

AD-A190 558

PLAYING POKER: A FULL PROGRAMMING EXAMPLE USING THE  
POKER ENVIRONMENT(U) WASHINGTON UNIV SEATTLE DEPT OF  
COMPUTER SCIENCE L SNYDER SEP 85 TR-85-09-02

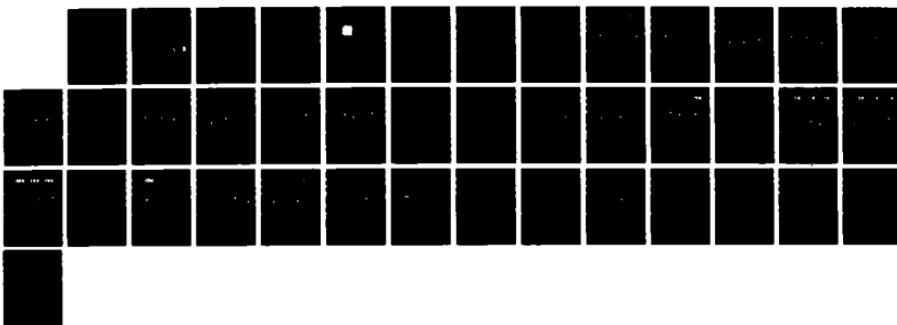
1/1

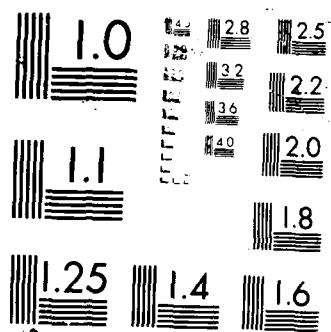
UNCLASSIFIED

NS0014-85-K-0328

F/G 12/5

NL





AD-A190 558

DTIC FILE COPY 3

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER none	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle)  Playing Poker: A Full Programming Example Using the Poker Environment		5. TYPE OF REPORT & PERIOD COVERED  Technical Report
7. AUTHOR(s)  Lawrence Snyder		6. PERFORMING ORG. REPORT NUMBER  N00014-85-K-0328
9. PERFORMING ORGANIZATION NAME AND ADDRESS  University of Washington Department of Computer Science, FR-35 Seattle, Washington 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS  Office of Naval Research Information Systems Program Arlington, VA 22217		12. REPORT DATE  September 1985
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		13. NUMBER OF PAGES  38
		15. SECURITY CLASS. (of this report)  Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)  Distribution of this report is unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)  S E D C E		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)  parallel programming		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)  This paper gives a complete illustration of a programming session with the Poker Parallel Programming Environment. The problem solved is to find the elementwise sum of a set of vectors, that is streams of data. The sample session serves two audiences: Those who wish to see an example of Poker without the specific details of a dry reference manual, and those who have Poker available online and who wish to gain operational proficiency.		

## **Playing Poker: A Full Programming Example Using the Poker Environment**

Lawrence Snyder

Department of Computer Science, FR-35  
University of Washington  
Seattle, Washington 98195

TR 85-09-02

NTIS GRA&I        
DTIC TAB        
Unannounced        
Justification

By \_\_\_\_\_

Distribution/  
Availability Codes

Dist	Avail and/or Special
A-1	

Abstract

This paper gives a complete illustration of a programming session with the Poker Parallel Programming Environment. The problem solved is to find the elementwise sum of a set of vectors, that is streams of data. The sample session serves two audiences: Those who wish to see an example of Poker without the specific details of a dry reference manual, and those who have Poker available online and who wish to gain operational proficiency.

This document has been funded in part by the Office of Naval Research Contract No. N00014-85-K-0328 and the National Science Foundation Grant No. DCR-8416878.

# Playing Poker: A Full Programming Example Using the Poker Environment<sup>1</sup>

Lawrence Snyder  
University of Washington

*The Purpose.* This document is intended to serve as a *sample program* for the Poker Parallel Programming Environment [1]. However, Poker programs do not exist in the traditional sense: There is no monolithic sequence of symbolic text. Rather, Poker keeps a "source database" which the programmer can view and change using interactive graphics. Databases and interactive graphics may be very convenient for the programmer, but they are not easily conveyed in hardcopy form. So, in an effort to provide some sense of the flavor of Poker, this document is composed mostly of annotated pictures produced in the course of a Poker programming session.

This example is intended to be as comprehensible as possible without requiring the reader to be fully familiar with the massive background material. The presentation is directed at two audiences: Those who do not have Poker available but would like to see specific details in a form that is not as dry as a reference manual, and those who do have Poker available and would like a quick way to gain operational proficiency. Both audiences can learn *how* things are done in Poker. For an explanation of *why* things are done as they are, two papers, written for general audiences, are available, one on Poker [1] and one on the CHiP architecture [2].

The text presented here refers to the Poker Programmer's Reference Guide [3] and it is convenient to have a copy available while following the example.

*The Problem.* We select a problem which is easy to understand and still illustrates many aspects of the system:

Given  $k$  vectors of length  $n$ , produce the  $n$  vector that is their elementwise sum and append to that resulting vector the grand total, i.e. the sum of all input values.

We will solve the problem with a binary tree based algorithm. Each vector is treated as a data stream entering a leaf of the tree. The leaf simply passes the value to its parent. Each internal node sums the pair of elements received from its children and passes the result to its parents. Additionally, the root accumulates all the results it sends out and after the result of the  $n$ th pair has been summed in and sent out, it passes the grand total out. For the first pass, we will use  $k=8$ ,  $n=10$ .

---

<sup>1</sup>The work described herein is part of the Blue CHiP Project and has been supported in part by Office of Naval Research Contracts N00014-84-K-0143 and N00014-85-K-0328 and NSF Grant DCR 8416878.

*The Display.* Figure 1 shows a typical Poker display with the relevant regions indicated.

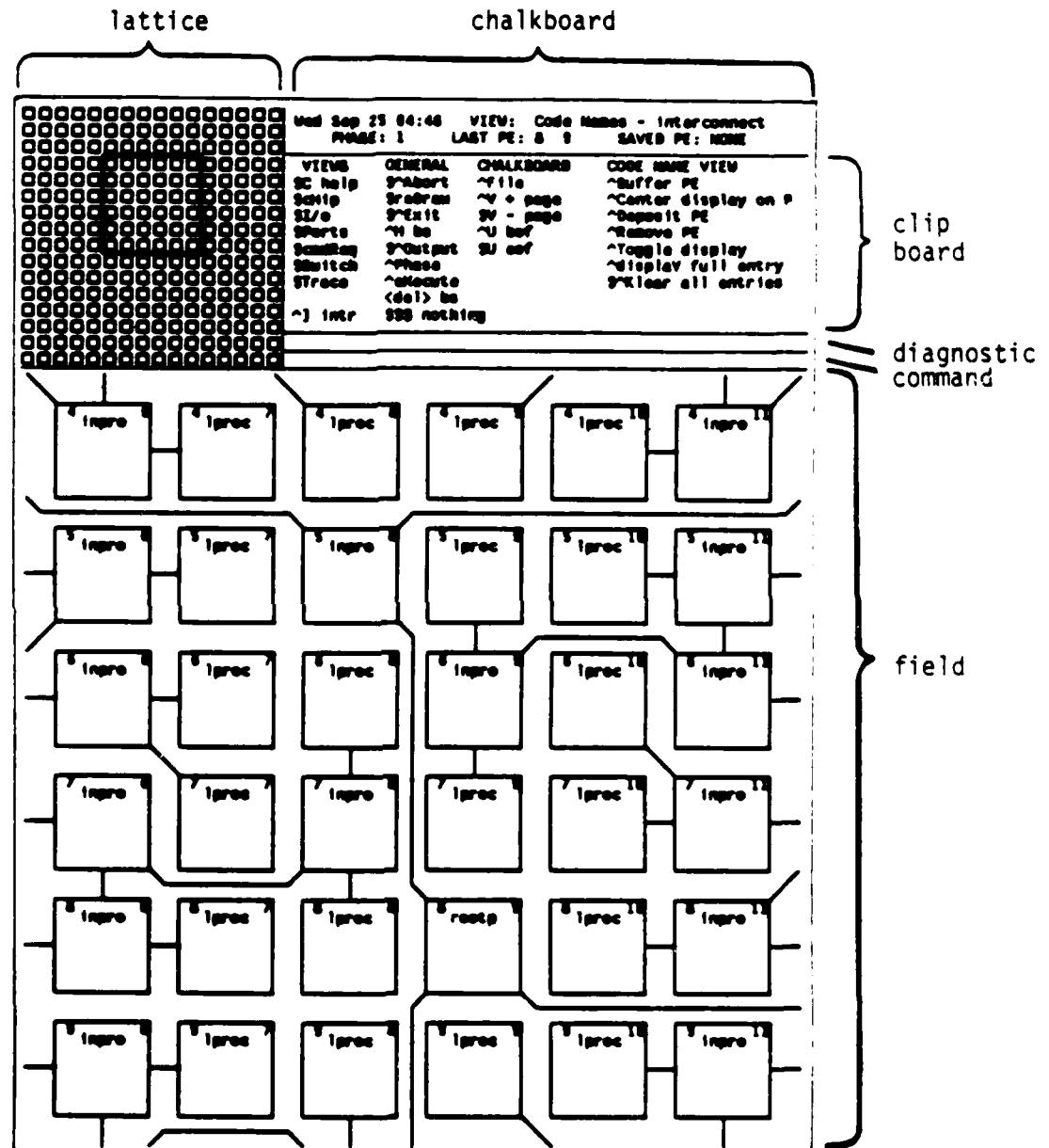


Figure 1. A Typical Poker Display.

The purpose of each region of the display is as follows:

<b>field</b>	A region showing a schematic picture of the CHiP Computer's lattice; this is the region where most programming activity takes place.
<b>lattice</b>	A schematic diagram of the processing elements (PEs) with a box enclosing those PEs currently shown in the field; no direct user activity is available in the region.
<b>chalkboard</b>	The upper righthand region of the display giving status information.
<b>command line</b>	The area where textual commands are given; last line of the chalkboard.
<b>diagnostic line</b>	The area where error indications are given; the next-to-last line of the chalkboard.
<b>clipboard</b>	A ten line region of the chalkboard used for the display of transient information and available for displaying files.

Notice that the display provides considerable geometric context which many users find helpful.

*The Keyboard and Keys.* In order to be completely precise, the example is annotated with every key struck by the programmer. To assist in making these strings intelligible, the accompanying legend (Figure 2) will be useful. Note that there is a keypad (Figure 3) for cursor motions. A keypad key prefixed by an escape (\$) is called a "gross cursor motion" since it moves a larger amount than the "fine cursor motions" which are unprefixed. Users having a MOUSE should refer to Appendix B of the Reference Guide[3] for alternate instructions.

\$	escape key
'<char>	denotes striking the <char> key while depressing the control key
%<char>	the <char> key of the keypad
b	blank; all spacing in command strings is for clarity

Figure 2. Legend of Poker Orthography

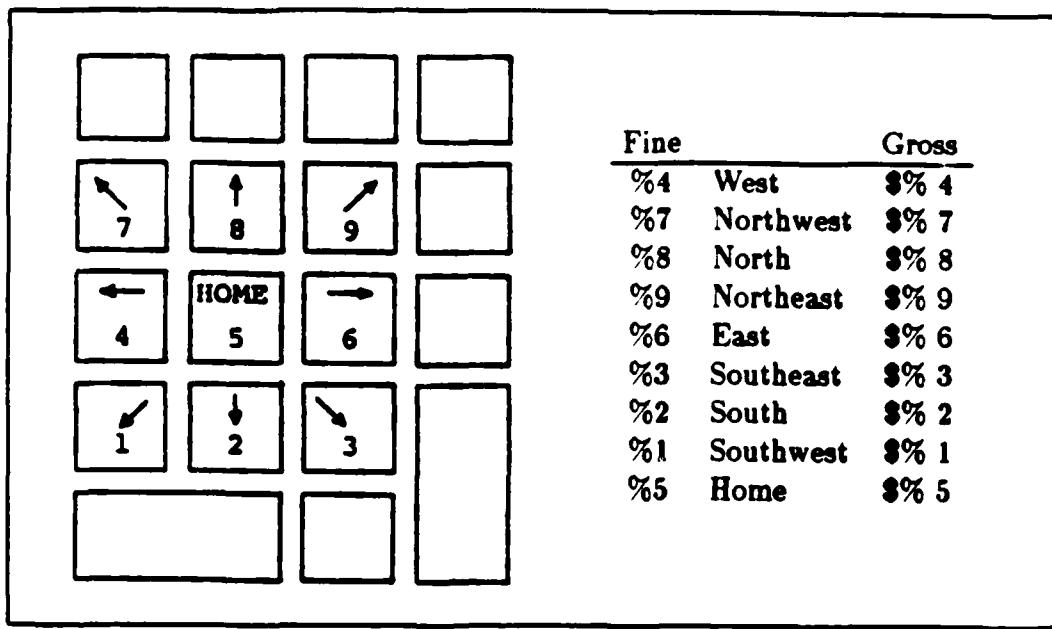


Figure 3. The Keypad of the Bitmapped Display  
and the Notation for Directional Keys

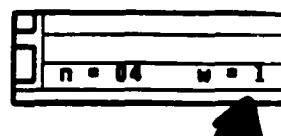
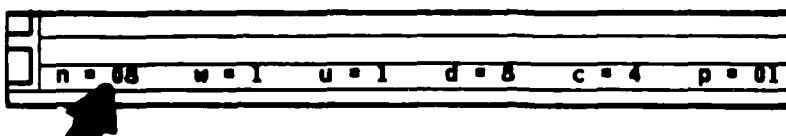
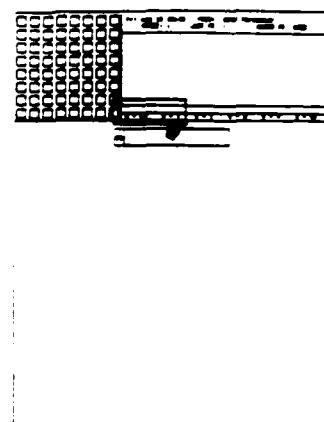
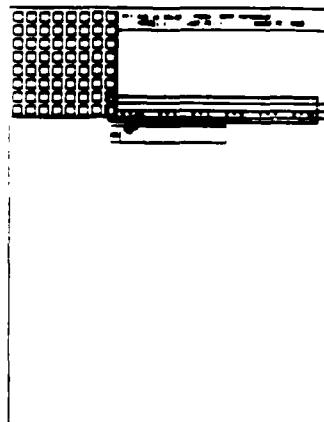
## Acknowledgments

Poker is the product of the ideas and efforts of many people. Janice E. Cuny and Dennis B. Gannon, in addition to contributing to the definition of the XX programming language, were a continual source of ideas, judgement and constructive criticism. Version 1.0 of Poker was written during the summer of 1982 by a delightful and committed group of gentlemen, the "Poker players": Steven S. Albert, Carl W. Amport, Brian G. Beuning, Alan J. Chester, John P. Guaragno, Christopher A. Kent, John Thomas Love, Eugene J. Shekita and Carleton A. Smith. Primary contributions to Version 1.1 were made by Steven J. Holmes and Ko-Yang Wang; their work steadily enhanced the system. The 1984 "Poker players", another congenial group, completed Versions 2.0 and 3.0: Kathleen E. Crowley, S. Morris Rose, James L. Schaad and Akhilesh Tyagi. Philip A. Nelson and David G. Socha assisted Kay Crowley and Jim Schaad during 1985 in producing Version 3.1. This document owes its existence to the hard work and good nature of Debra Sanderson and Eriko De La Mare. It is a pleasure to work with such fine people and to acknowledge these valuable contributions.

## **References**

- [1] Lawrence Snyder  
The Poker Parallel Programming Environment  
*Computer*, 17(7):27-36, July 1984.
- [2] Lawrence Snyder  
Introduction to the Configurable Highly Parallel Computer  
*Computer*, 15(1):47-56, January 1982.
- [3] Lawrence Snyder  
The Poker (3.1) Programmer's Reference Guide  
Technical Report 85-09-03, University of Washington, 1985

```
bluechipx mkdir sample  
bluechipx cd sample  
bluechipx ls  
bluechipx poker
```



1. Once you are logged into UNIX from a bitmapped display and referencing a clean directory, type 'poker' to start the system. If this does not work, either Poker is not available on your machine or your PATH line in the .profile or .login file does not refer to the directory containing Poker, e.g. /usr/poker/bin.

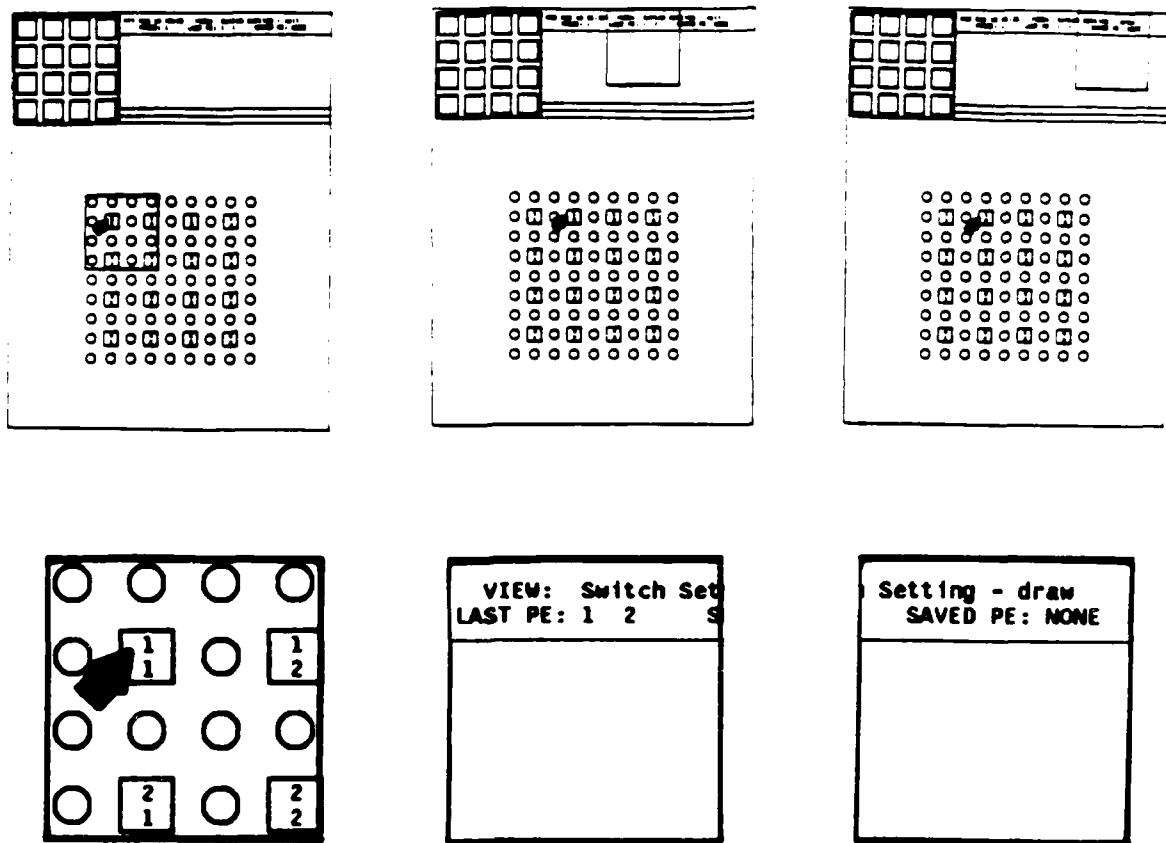
In subsequent figures, arrows illustrate the cursor position.

2. Poker begins in CHiP Parameter view which is used to declare the characteristics of the CHiP computer being programmed. The default architecture's parameters define a 64 processor machine since n, the number of processors on the side of the  $n \times n$  lattice, is given as  $n=64$ . We need only a 16 processor machine for our example so we must change the entry for n to be 4 by moving the cursor east one character and striking 4.

[%6 4].<sup>1</sup>

3. Changes to the CHiP parameters do not take effect until we move to some other view, so we move to Switch Settings[\$]. (Discussion of the meaning of the other CHiP parameters is given in Section 7 of the Reference Guide.)

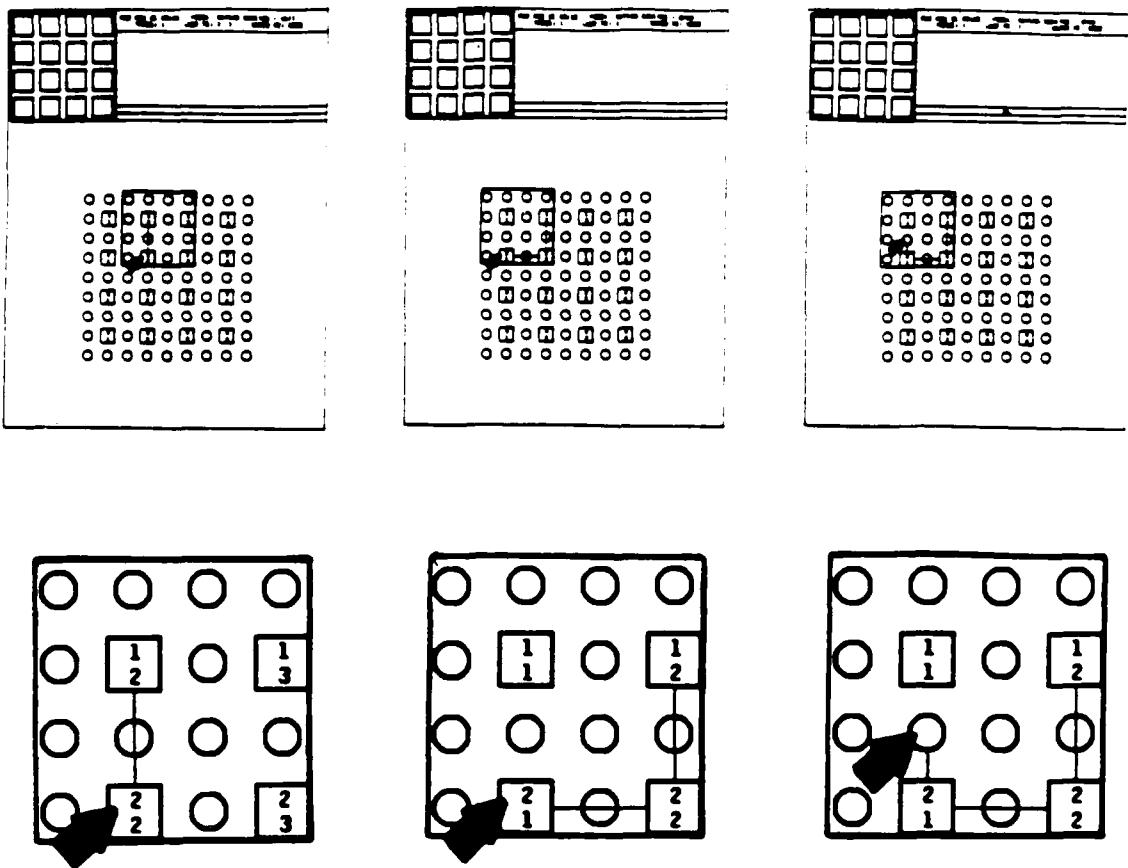
<sup>1</sup>Recall that %(char) refers to the (char) on the key pad at right, \$ is escape, ^ is control, and & is a required blank.



4. To specify the communication graph connecting the processor elements (PEs) for our problem, we draw a picture of a binary tree. The 16 PEs are shown as boxes with indices; the circles are switches through which all communication paths (lines of the picture) pass. The root of our binary tree should be at PE 1 2 so we move the cursor east. This can be done either with two fine cursor motions: [%6 %6] or one gross cursor motion [8%6] which is given by prefixing the keypad key with an escape. (A discussion of other cursor manipulation activities is given in Section 3 of the Reference Guide.)

5. An indication is given in the chalkboard of the index of the last PE referenced. To draw the communication paths in the lattice, we set the cursor mode to "draw" ['d] and then move the cursor around the lattice.

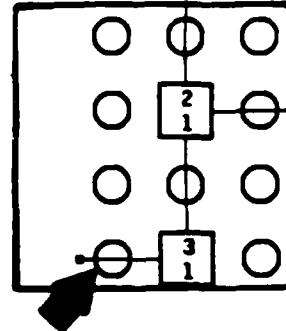
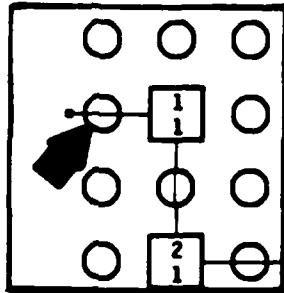
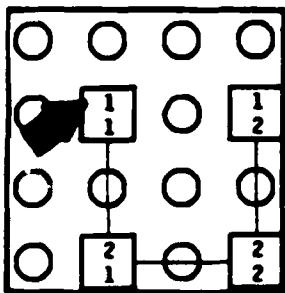
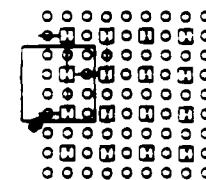
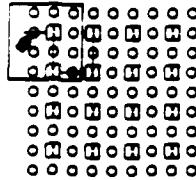
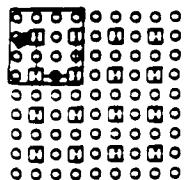
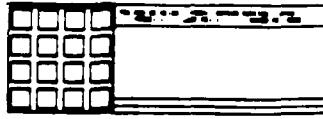
6. The "draw" indication replaces the "null" (no mode set) indicator in the chalkboard. We next move the cursor south [8%2] to the child of the root, i.e. PE 2 2.



7. The communication path between the root and its left child - its right will be PE 2 3 - is shown as a line. The processors will be able to communicate values over this link. Since "drawing" is still in effect, we can move west to the (root's) grandchild [9%4].

8. The communication path between child and grandchild is shown as before. Gross cursor motions move to the next PE in the indicated direction; on our way to the great-grandchild, we can illustrate fine cursor motions, which move to the next element, be it a switch or PE [%"].

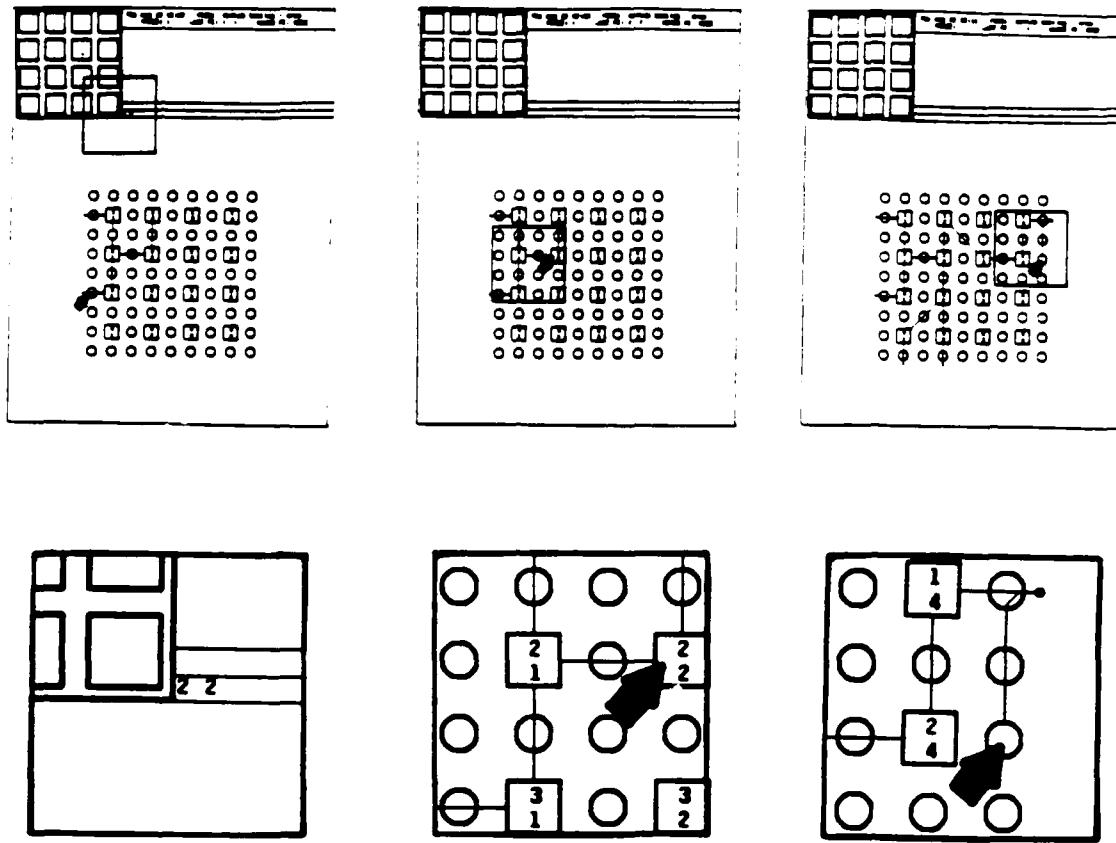
9. Notice that the line stops at the edge of the switch. (Communication paths can enter and leave a switch from the eight compass points, and they can cross over one another. See Section 8 of the Reference Guide.) We move to PE 1 1, a leaf [%8].



10. Having reached a leaf, we want to connect to a stream of data values stored on the disk. Such streams "enter" the lattice through "pads"; pads are indicated by tiny squares "outside" the lattice and they are created by moving the cursor "off" the edge of the lattice [%4].

11. We can draw the edges to the sibling leaf, i.e. PE 3 1, by retracing our path back through PE 2 1 [%6 %2 %2 %4].

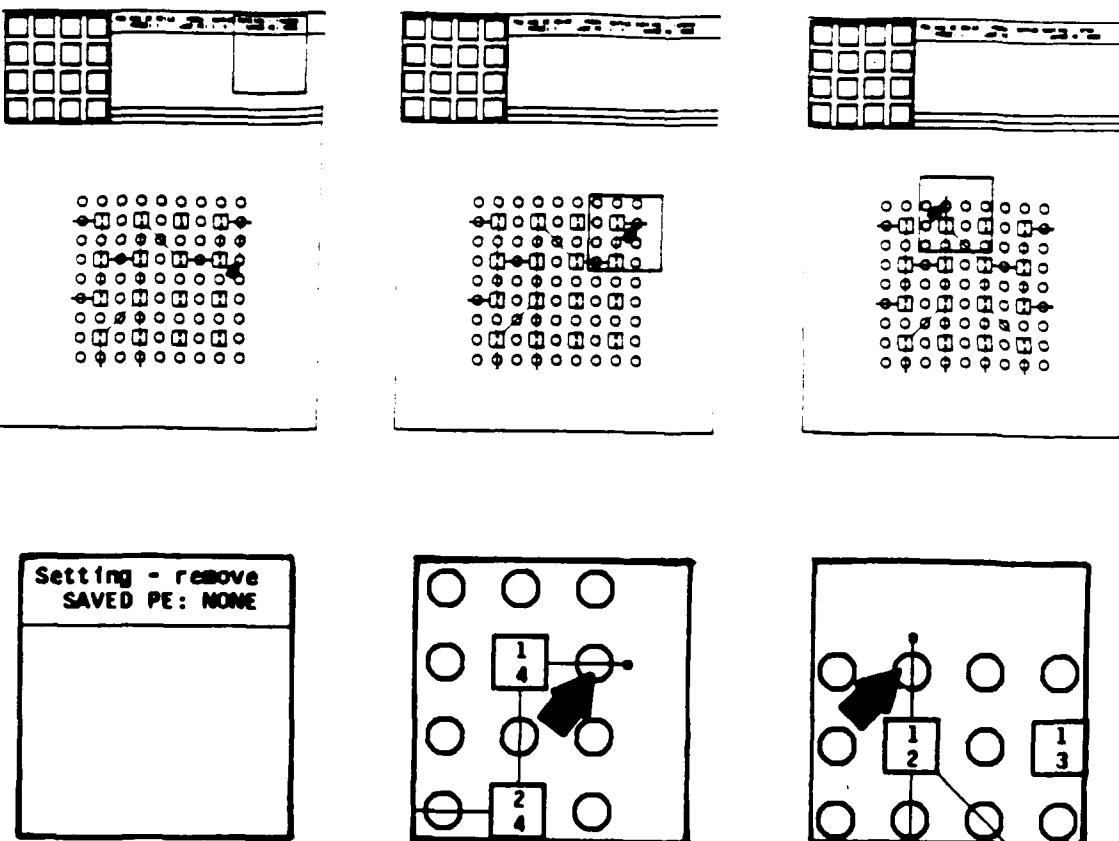
12. To continue the construction of the tree we must return to PE 2 2, the child, and this could be done by retracing the path back to PE 2 2, or by turning off the "draw" indicator (e.g. with 'n) and moving the cursor to PE 2 2. A third possibility is to use the Center command. We begin by typing the index of the PE, [2b2]; the characters will be displayed on the command line even though the cursor remains in its present position.



13. With the PE index given on the command line, Center [^c] will move the cursor to the PE.

14. We continue in an analogous way to define the binary tree and its pads [\\$%2 \\$%1 \\$%2 \\$%8 \\$%9 \\$%2 \\$%2 \\$%8 \\$%8 \\$%8 \\$%8 \\$%3 \\$%6 \\$%8 \\$%6 \\$%2].

15. At this point we have made a mistake which must be corrected. This is done by changing the mode from "draw" to "remove" [^r].



16. The change, reflected in the chalkboard, allows us to "backup" over the line, removing the setting [\\$%8].

17. With the arrow corrected, we can reset the indicator to "draw" and continue with the graph layout [^d] and continue with the graph layout [\\$%4 \\$%2 \\$%2 \\$%6 \\$%4 \\$%8 \\$%4 \\$%2 \\$%3 \\$%2 \\$%8 \\$%7 \\$%2 \\$%2 1b2 ^c \\$%8].

18. The layout of the binary tree into the lattice is completed: Eight data streams enter at the leaves of the tree and a result stream exits at the root. The completed switch definition is given in Figure 4. To assign processes to each of the processors, we change to the Code Name View [Sc].

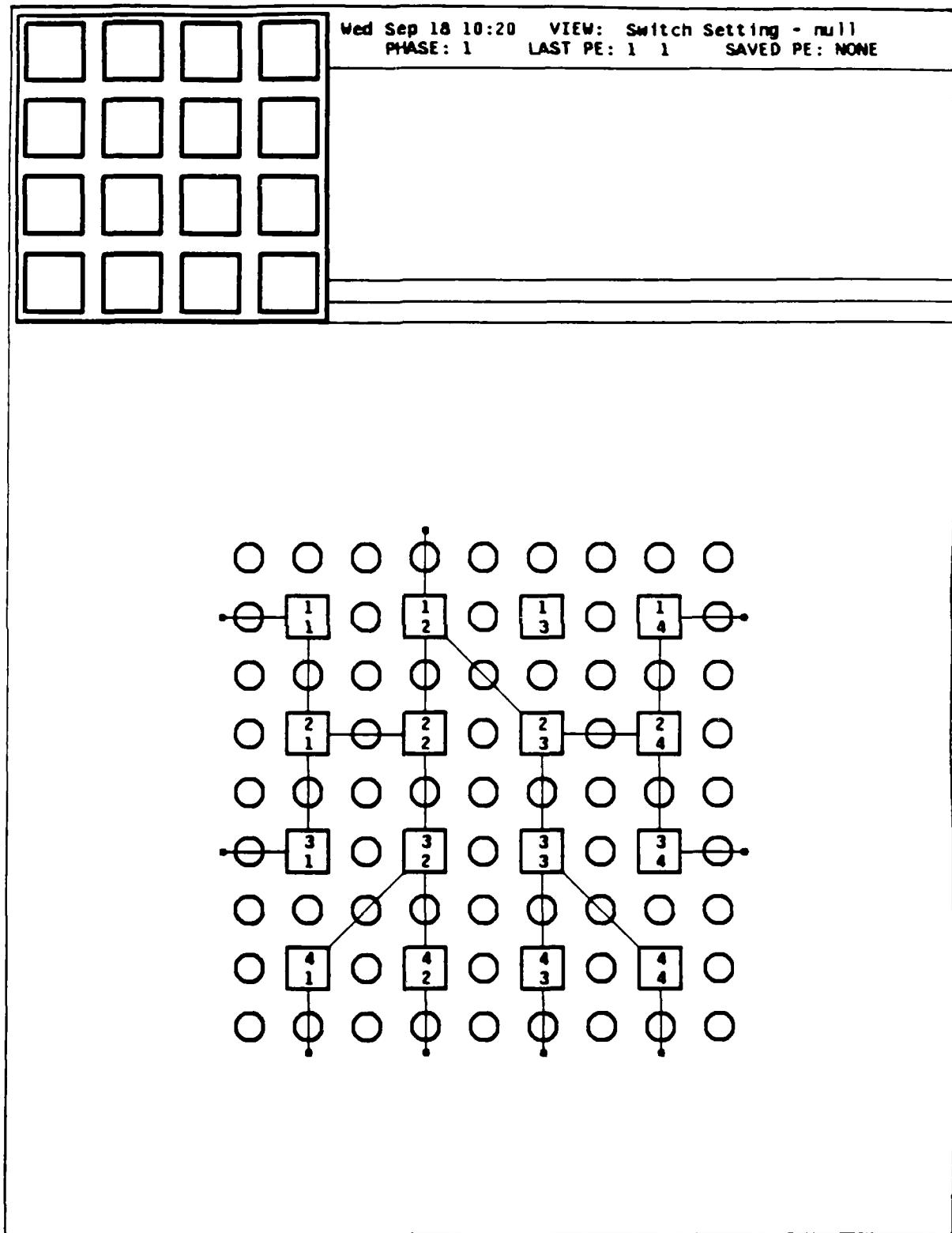
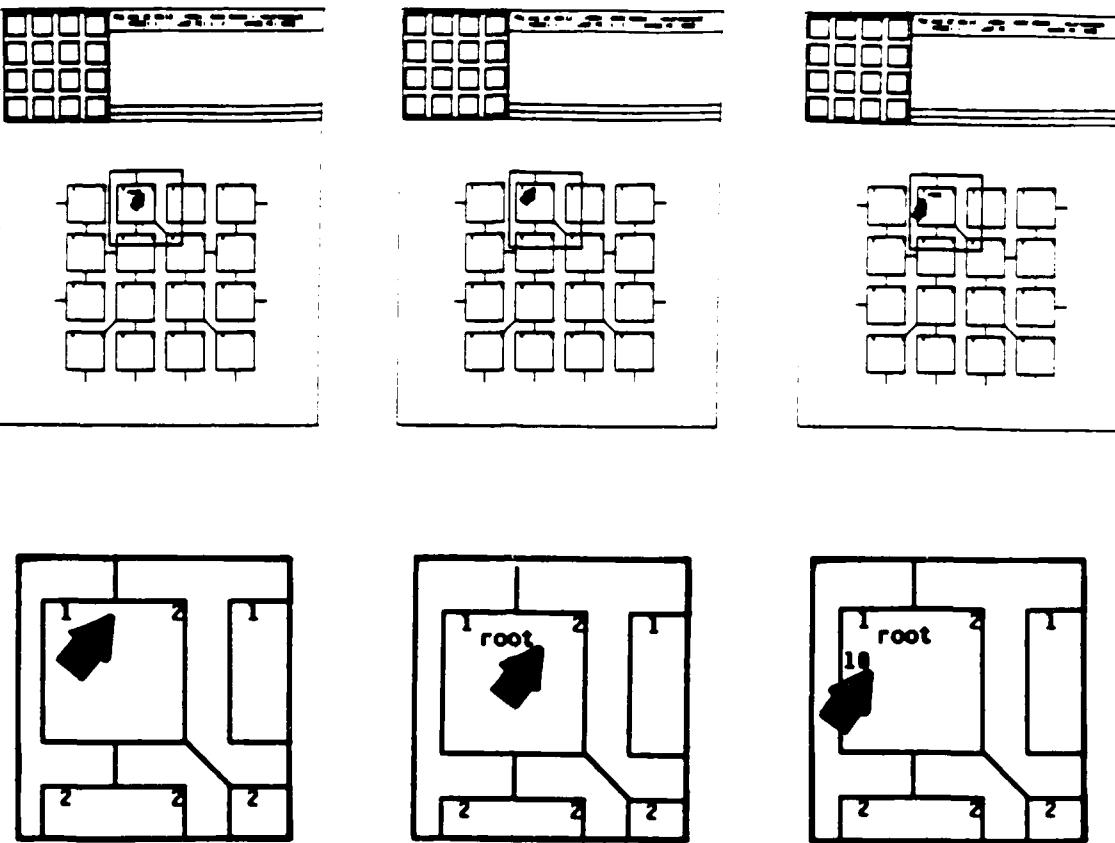


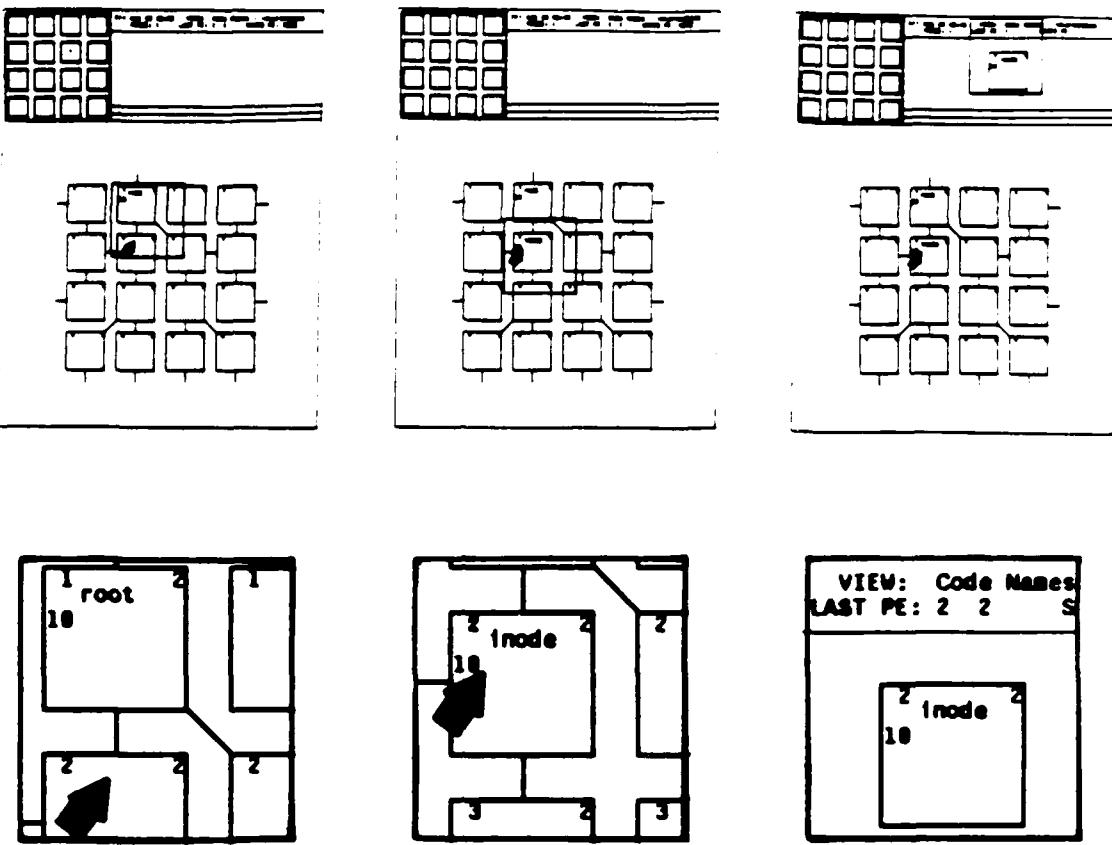
Figure 4.



19. Notice that the switches have been removed and the PE boxes enlarged to hold more information. In Code Names View we assign processes to processors by entering into the PE boxes the names of the (yet to be written) routines. The window for the code name is centered one line down from the top of the box, so we move down to it [%2] and enter the text [root].

20. The name (of up to 16 characters) is clipped to the first five characters. The four remaining lines of the PE box are also visited by fine north/south cursor motions and are used for specifying (actual) parameters to the process. Since we intend for the stream length to be a parameter, we enter its actual value now [%2 10].

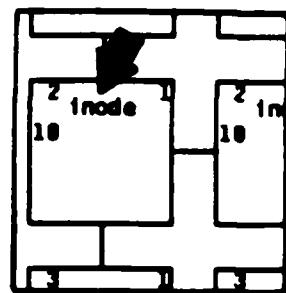
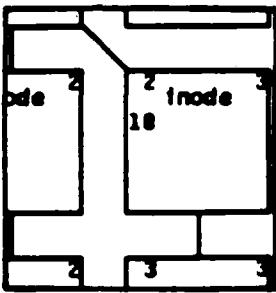
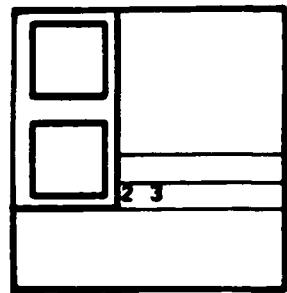
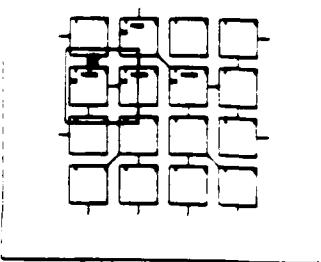
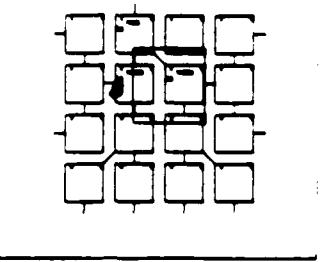
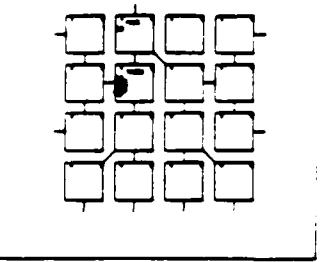
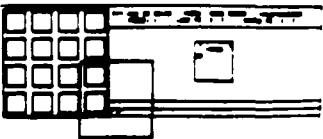
21. As before gross cursor motions move from PE to PE, so we move to the child PE[%2]; the cursor will appear at the top of the box in what is called the "home" position.



22. To enter its process code name in the appropriate window, we move down to the first line [%2]. The interior nodes will be given the "inode" process name and the stream length actual parameter [inode %2 10].

23. Since there are several PEs that will receive this same text, it is convenient to buffer the entries ["b]. The buffered cell is shown in the clipboard region of the chalkboard.

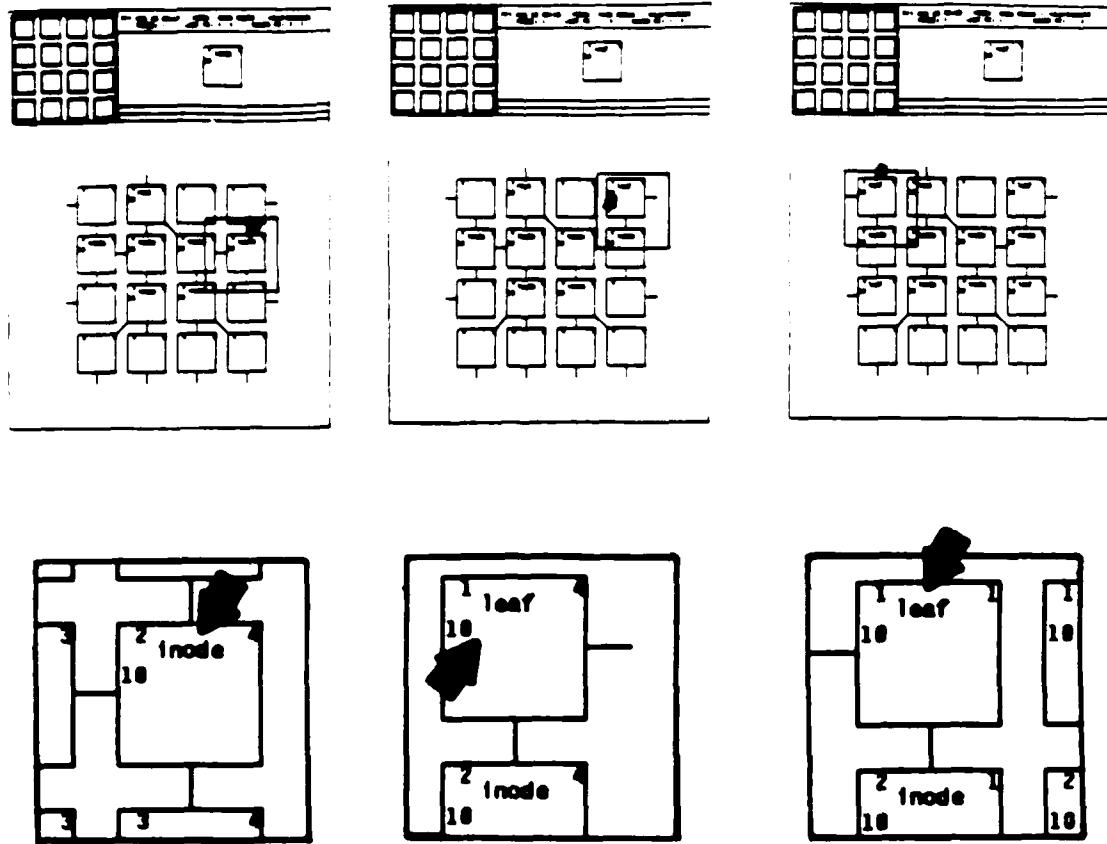
24. To deposit the buffered values in another PE, say the sibling PE 2 3, we can type the index of the recipient PE; but to avoid the possibility of the index specification being treated as text, we must move out of the window and to the 'home' position of the PE [%5]. Then we enter the indices [2&3] which are shown on the command line.



25. With the recipient indices specified, a deposit command [`^d`] copies the buffered values into the cell. (The cursor does not move as a result of the deposit. For a full discussion of the buffer-deposit mechanism, see Section 9 of the Reference Guide.)

26. Another way to use the deposit feature is to move the cursor to the recipient PE and invoke the deposit, because when no PE indices are specified on the command line, the deposit is to the PE containing the cursor. To illustrate this case, we move to PE 2 1, the grandchild [`$%4`], and execute a deposit [`^d`].

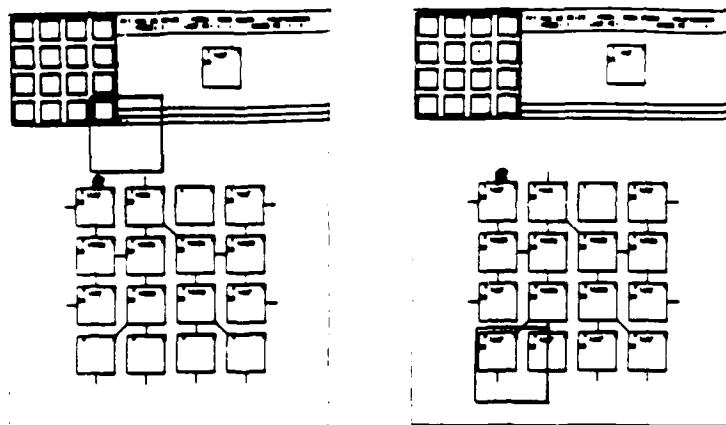
27. Similar motions can easily specify all of the internal node processes [`$%3 ^d $%6 ^d $%9 ^d`].



28. Next we visit the leaf at PE 1 4 and assign the leaf code and actual parameters [\\$%8 %2 leaf %2 10]. In preparation for subsequent specifications, we buffer the text (^b).

29. We move to the sibling leaf (PE 3 4) [\\$%62 \\$%62] and deposit the text (^d). Then we continue to deposit leaf code assignments to all leaves above the last row [\\$%4 \\$%4 \\$%4 ^d \\$%8 \\$%8 ^d].

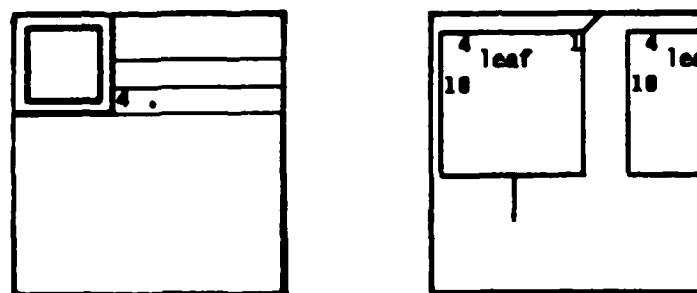
30. With the whole fourth row requiring the same text, we define an iteration [4..] which specifies a group of PEs: Their first index is 4 and their second index, specified as a period, means any legal value, i.e. period abbreviates 1..4 in this case. The iteration is shown on the command line.



```

code leaf(n);
trace x;
ports in,parent;
begin
  int x,i,n;
  for i := 1 to n do
    begin
      x <- in;
      parent <- x
    end
end.

```



31. A deposit [^d] assigns the buffered text to all of the fourth row PEs. The binary tree's processes are now assigned. (See Figure 5.)

32. The next step is to define the sequential code for each process. This uses the XX programming language and is usually done on the companion terminal using a standard editor. (If so, the terminal should refer to the *sample* directory; if a companion terminal is not available, the user can exit Poker (\$ ^c) to prepare the processes.) The process code is given one process per file with the naming convention <process name>.x. (To save typing, these files can be found in /usr/poker/lib/Playing Poker.) We give the XX codes for the three tree processes.

33. The leaf process, stored in the file leaf.x, simply reads a value from its input port, in, and writes it to its parent. Notice that data is transferred one value at a time using an assignment-like operator (<-). (More information on the XX language is given in Section 12 of the Reference Guide.)

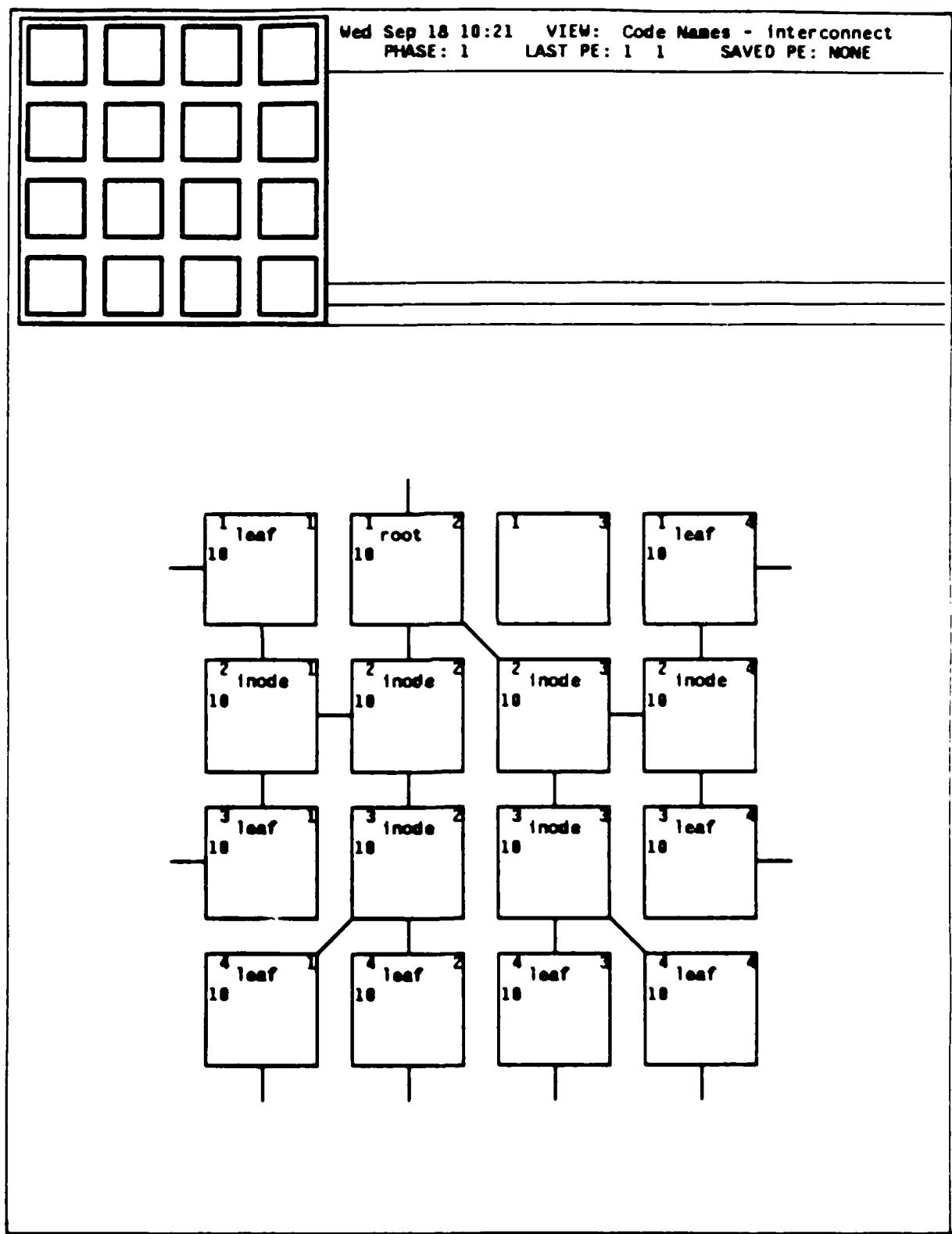


Figure 5.

```

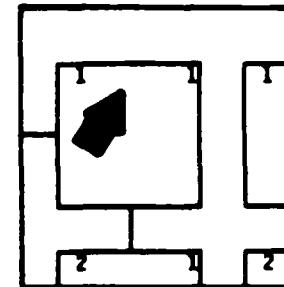
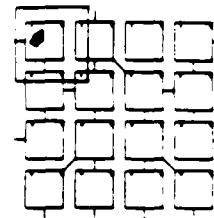
code inode(n);
trace z;
ports lchild,rchild,
parent;
begin
int x,y,z,i,n;
for i := 1 to n do
begin
x <- lchild;
y <- rchild;
z := x+y;
parent <- z
end
end.

```

```

code root(n);
trace w,z;
ports lchild,rchild,
parent;
begin
int w,x,y,z,i,n;
w := 0;
for i := 1 to n do
begin
x <- lchild;
y <- rchild;
z := x+y;
parent <- z;
w := w + z;
end;
parent <- w
end.

```



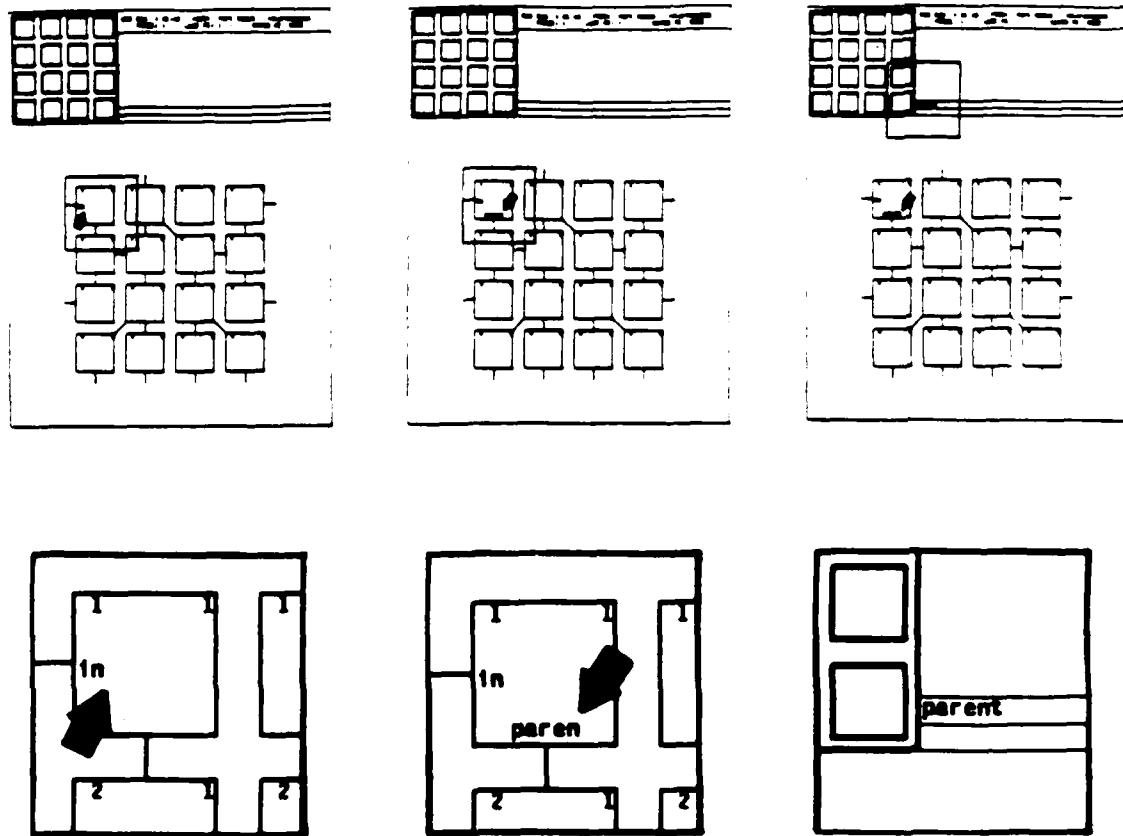
34. The internal nodes read values from their child processes, sum the values received, and send the result to their parents.

35. The root acts like any internal node, but it also retains a running total (w) which it appends to its output stream at the end of the computation.

Having referred to neighboring processes by port names, we must now define them.

36. The Port Names View is entered [Sp] showing a display similar to that of Code Names. The chief difference is that the PE box is broken up into eight windows, one for each compass point. The windows contain names of at most 16 characters which are clipped to the first five. The fine cursor motions move to the windows; gross cursor motions move between PE boxes.

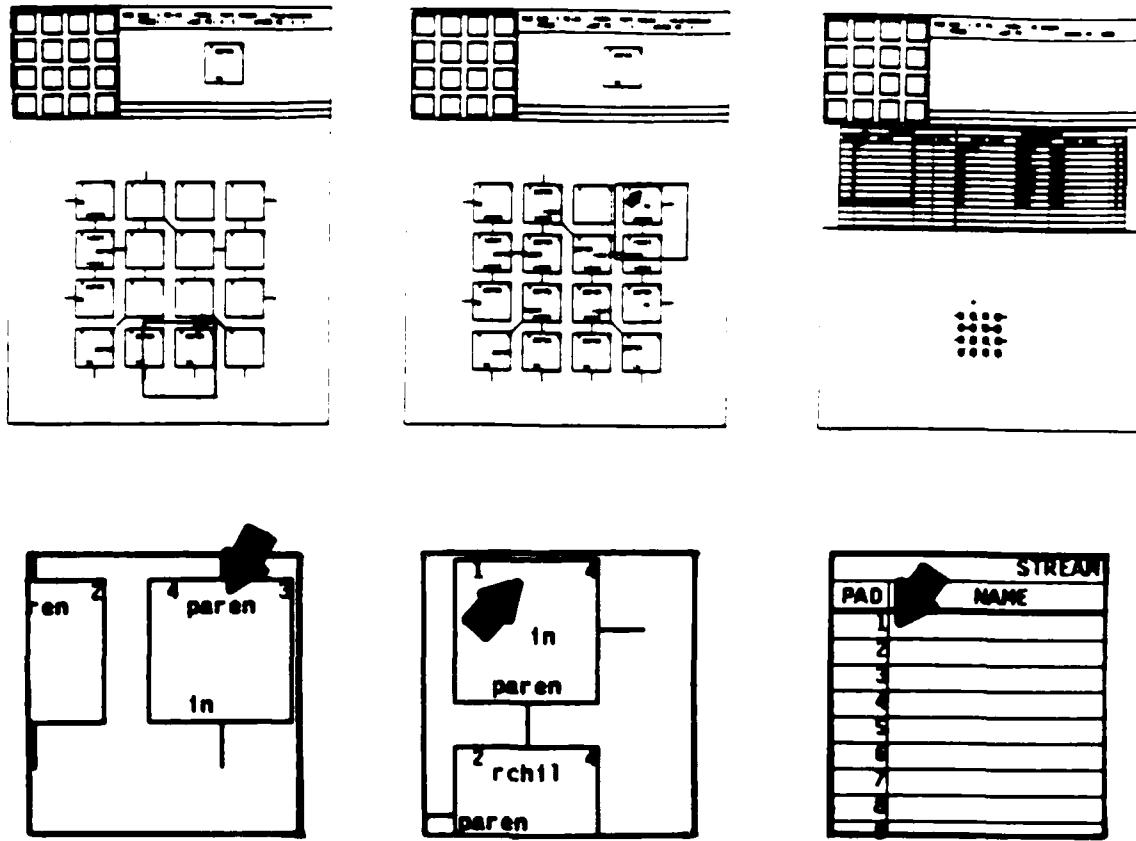
I	(-n-)
(nnn)	(-ne)
(-w-)	(-e-)
(sw-)	(-se)
(-s-)	



37. Using the fine cursor key, we move to the west port window [%4] and enter the port name used in the leaf processes for the external data stream [in]. Similarly, we can move to the south window [%2] and enter the other port name [parent].

38. The windows are small (5 characters) although the actual names can be up to 16 characters in length. To see the unclipped entries we can use the display key [^y]. The result is shown on the diagnostics line.

39. We can continue making entries by using gross motions to move to new PEs [%2] and then fine motions to move to the port windows [%8 lchild %6 parent %2 rchild]. Continuing in this way [%2 %8 parent %4 in %2 %9 parent %2 in %8 %8 parent %2 in] we complete a cell that can be buffered [^b] and reused [%8 ^d].



40. The irregularity of the embedding prevents us from making much use of the deposit feature, but this is to be expected: It is exactly the port naming facility that allows us to remove the geometric dependences of the layout and treat it in our programs as a logical tree. So we move around the lattice making the entries [\\$%7 %8 parent %1 lchild %2 rchild %3 %4 parent %4 lchild %2 rchild %3 %8 parent %2 lchild %3 rchild %3 %7 parent %2 lchild %6 rchild %2 %8 parent %2 lchild %3 rchild %3 %7 parent %2 in %8 %8 parent %6 in %8 %4 parent %2 lchild %8 rchild %3 %2 parent %6 in]. (See Figure 6.)

41. The ports allow us to name the processes' neighbors; now we must name the streams that flow through the pads. For this, we move to the IO Names View [§i].

42. The IO Names View is used to assign stream names to the external I/O pads that were defined in Switch Settings View. The diagram at the bottom of the field shows a schematic of the lattice. A table of stream name definitions is given at the top of the field; the user fills in the table entries for the stream; the remaining information is provided by Poker. To assign input streams first, we move the cursor to a leaf pad; to demonstrate that the pads are visited cyclically, we move up [%8] to wrap around to the last pad, leaf 1.

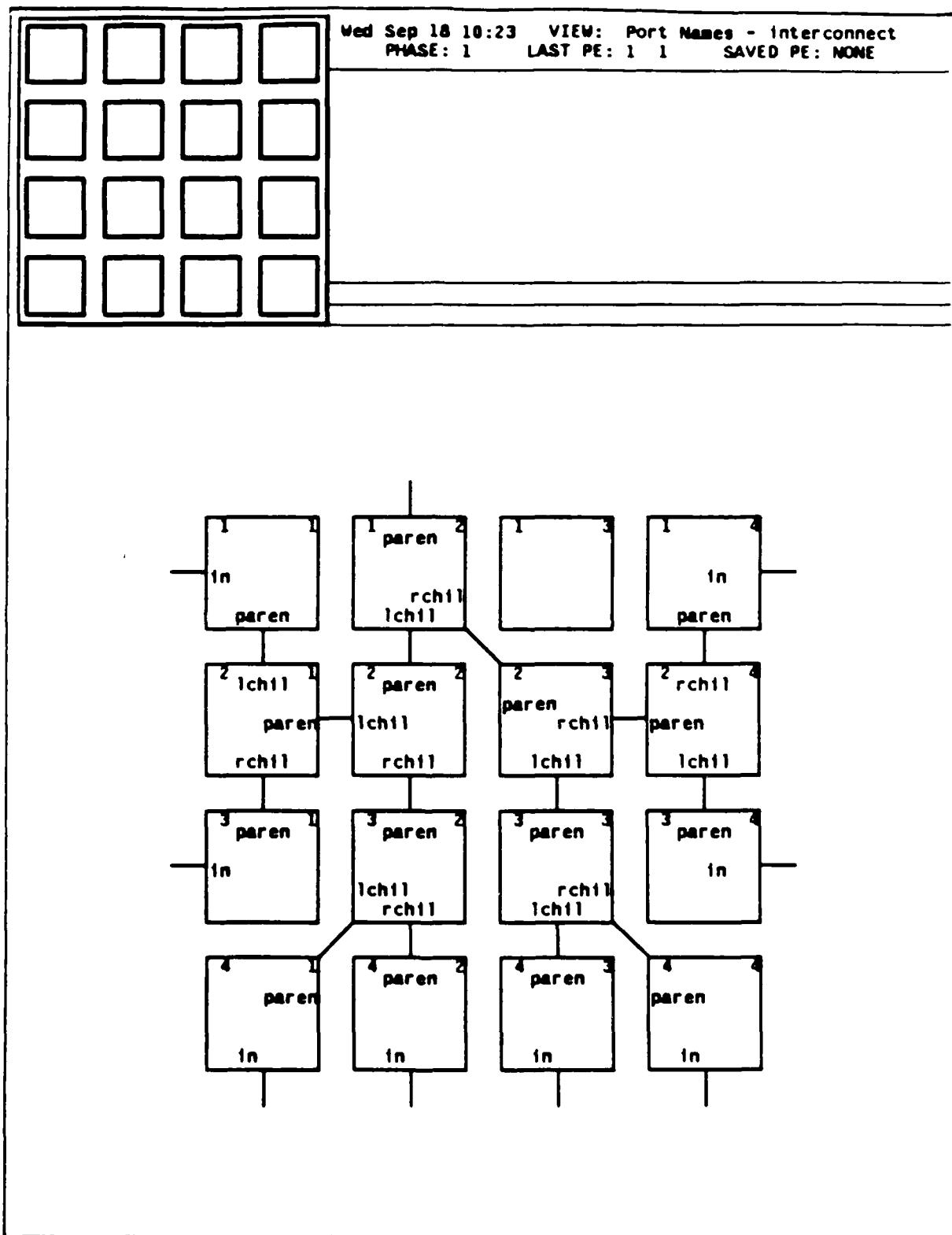
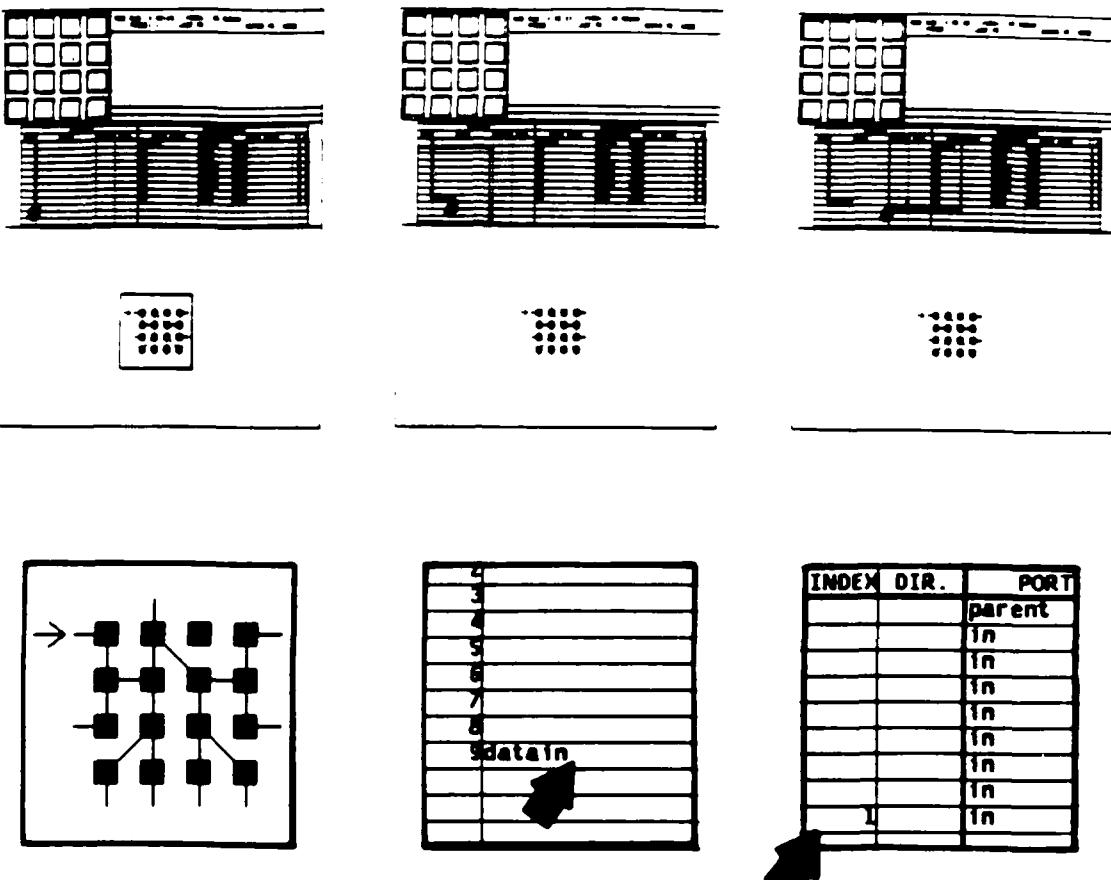


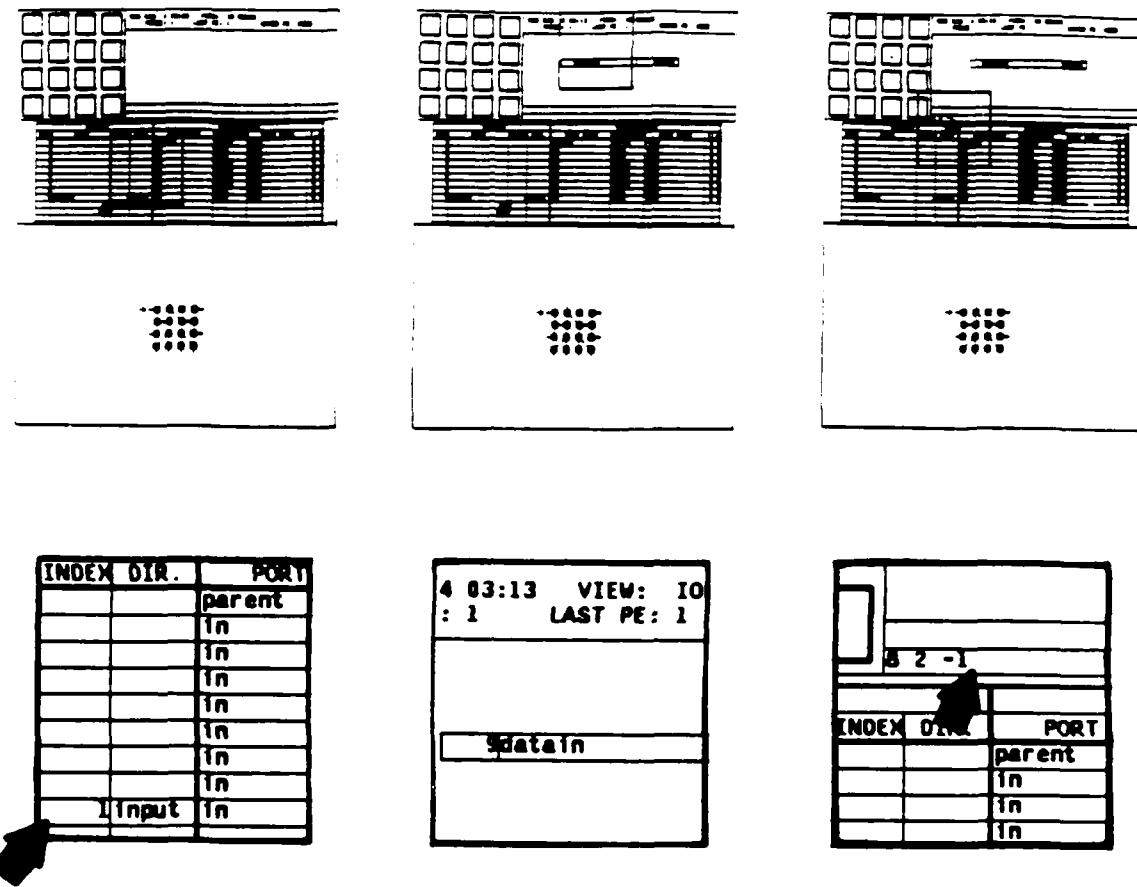
Figure 6.



43. Notice that the arrow in the schematic changed position to point to the position of the last pad. Pads are numbered clockwise starting from the northwest corner. To define the stream name "datain", we simply type into the "name" window pane [datain].

44. Streams, sequences of data values, are either files by themselves or they are grouped into a file by being placed "side-by-side". The index describes which position in the file the stream occupies, i.e. which field it is in each record. Assuming the leaves are numbered in the normal "left to right" order, pad 9 corresponds to leaf 1 and thus, we assume, to stream 1. So we move east to the index pane of the pad window [%6] and enter the index [1].

45. The direction refers to whether this stream is an input or output stream. Streams can only be unidirectional. We specify input [^i].



INDEX DIR.	PORT
	parent
	in
lInput	in

4 03:13 VIEW: IO  
: 1 LAST PE: 1  
  
\$datain

INDEX DIR.	PORT
	parent
	in
	in
	in

46. Since all of the leaves will have essentially the same entries (except for index), we can buffer this entry [^b] in anticipation of depositing it in the other leaf positions with an iterated deposit.

47. To give the iteration parameters for the iterated deposit command, we move to the command line using the "home" key [%5]. Then we give the first element of the iterated sequence, the last element, and the amount by which the step is to change, i.e. by -1 [862b -1].

48. The multiple pad deposit can now be specified with a single key [^d]. (A full description of iterated deposit is given in Section 11 of the Reference Guide.)

The figure consists of three horizontal panels. The top panel shows two tables: 'PAD' on the left and 'STREAM' on the right. The bottom panel shows a command line input field followed by a table.

PAD	NAME
1	
2	3datain
3	3datain
4	4datain
5	5datain
6	6datain
7	7datain
8	8datain
9	9datain

PAD	NAME
1	dataout
2	3datain
3	3datain
4	4datain
5	5datain
6	6datain
7	7datain
8	8datain
9	9datain

IN	INDEX DIR.	PO
	loutput	paren
8	8input	in
7	7input	in
6	6input	in

49. Notice that unlike deposits in the other views, which copy the buffered data exactly, deposit in IO Names updates the index by 1 as it deposits. This incrementing property explains why we iterated through the leaves in the order in which their streams appear in the file. Next we go back to the output pad [ %5 %2] using wrap around and enter the stream [dataout %6 1 ^o]. This completes the IO Names specification; the final form is shown in Figure 7.

50. Having completed the source specification of our example program, we take the precaution of making a backup copy of our database. This is done by moving to the command line [%5] and specifying the "copy" command [copy b]. (This is a general file manipulation facility.) The appropriate transfer for backing-up is to copy all internal source database entities [\*] to the current directory [.]. (File management is discussed in Section 6 of the Reference Guide.)

51. All execute commands are invoked with the execute key ['x]. (In the case of "copy", the completion is reported.) Other execute commands perform text substitution (replace), associate streams with files (bind), and execute programs (run). A list of the available commands is given when one "executes" a blank command line. When one executes a parameterless command, the syntax for that command is given. Incidentally, in a view the command for that view gives a list of the legal key strokes.

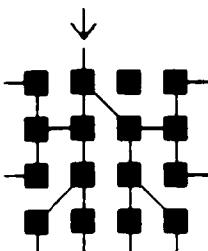
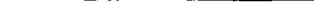
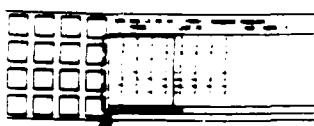
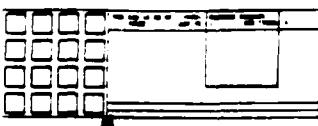
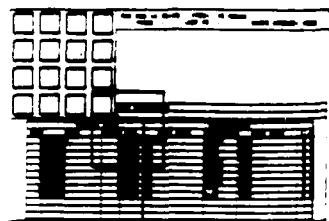


Figure 7.



copy process	
IN	DIR.
loutput	parent
ginput	in
ginput	in
ginput	in

Command Request  
1 1 NUM TICKS

1, 6, /, /,  
3, 3, 5, 5,  
6, 7, 1, 8,  
1, 3, 4, 2,  
-1, -1, -1, -1,  
-1, -6, -7, -7,  
-3, -3, -5, -5,  
-6, -7, -1, -8,  
-1, -3, -4, -2,  
1, 1, 1, 1,  
At beginning of /

52. Having completed the source database specification for our program, we move to the Command Request view where we prepare to run the program [Sr].

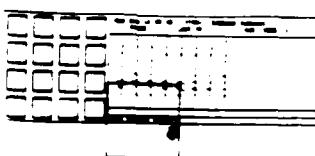
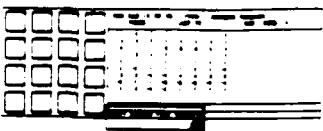
53. First we display the file of sample data that will be used to test the program. Files are displayed by giving the file name on the command line followed by ^f. The file is printed in the Clipboard region of the screen with the lines clipped, if necessary [/usr/poker/lib/Playing Poker/test ^f].

54. To be used by Poker, the file must be copied into the current directory. Since this is a file rather than a Poker database entity, we use the UNIX copy command rather than the Poker copy command. UNIX commands are executed from Poker using the "shell" execute command with the UNIX command as a parameter; thus, we type "shell" followed by the text UNIX is to execute followed by ^x. [shell b cp b/usr/poker/lib/Playing Poker/test b.^x]

```

1,   6,   7,   7,   7,   2,   1,   6,
3,   3,   5,   5,   4,   4,   3,   2,
6,   7,   1,   8,   8,   8,   6,   4,
1,   3,   4,   2,   1,   7,   7,   2,
-1,  -1,  -1,  -1,  -1,  -1,  -1,  -1,
-1,  -6,  -7,  -7,  -7,  -2,  -1,  -6,
-3,  -3,  -5,  -5,  -4,  -4,  -3,  -2,
-6,  -7,  -1,  -8,  -8,  -8,  -6,  -4,
-1,  -3,  -4,  -2,  -1,  -7,  -7,  -2,
1,   1,   1,   1,   1,   1,   1,   1,

```



**shell packIO test vectors**

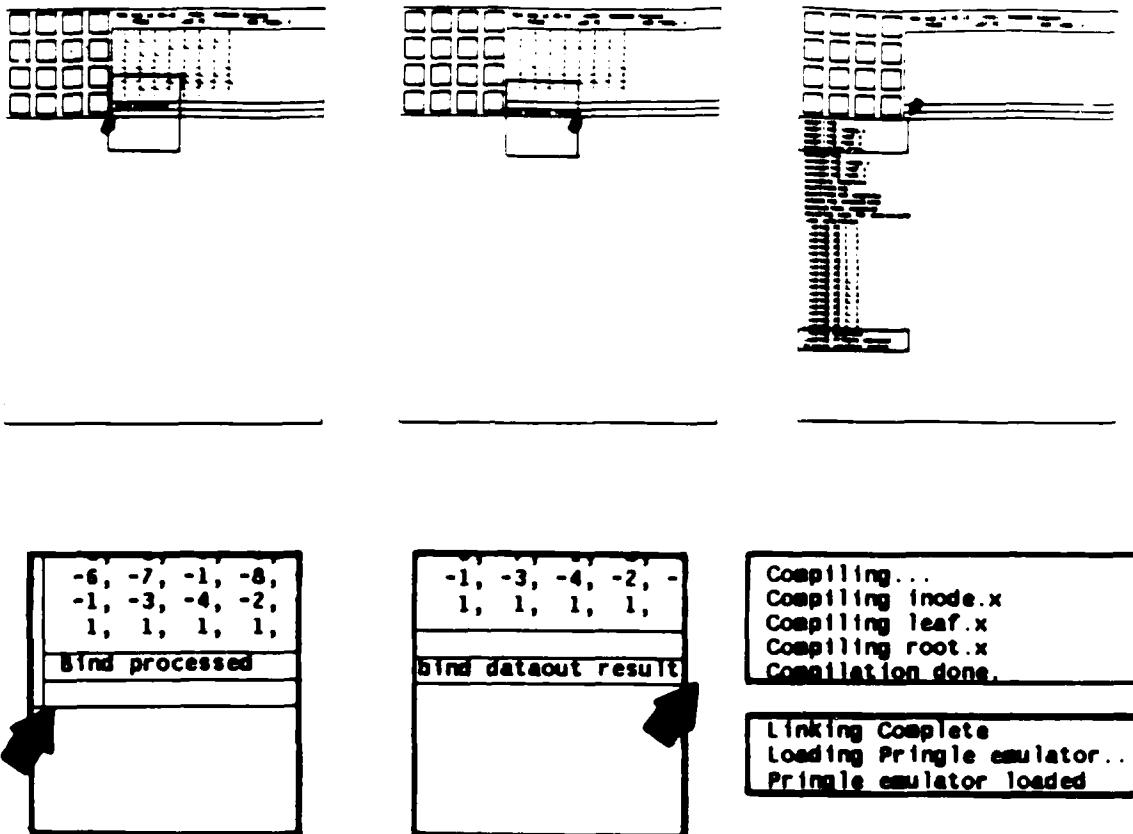
-6, -7, -1, -8, -  
-1, -3, -4, -2, -  
1, 1, 1, 1,

**bind datain vectors**

55. The file, now in the current directory, is composed of random values and their negations, so they sum to 0. It has 10 records of 8 fields each, and can therefore be interpreted as 8 streams (laid out side-by-side) each containing 10 elements.

56. Poker expects these streams to be in a special format which is produced by a utility program, called packIO, described in Appendix B. We use the shell command again to format the file [shell & packIO& test& vectors ^x] and name the result vectors.

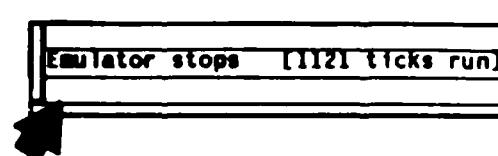
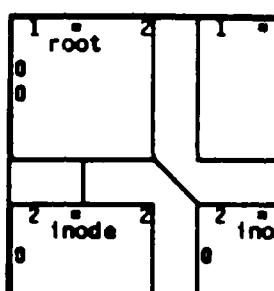
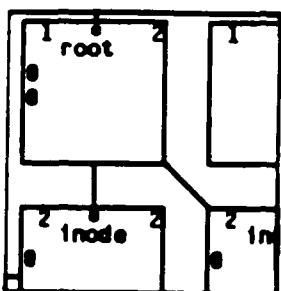
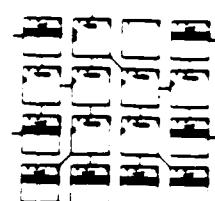
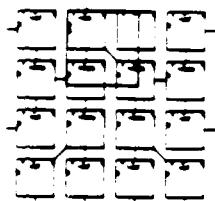
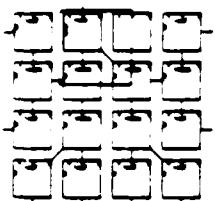
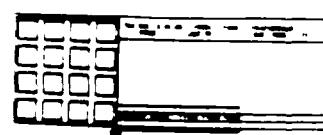
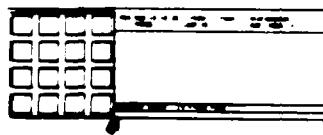
57. To associate the 8 streams, vector 1, vector 2, ..., vector 8 with the 8 streams declared in the IONames view, datain 1, datain 2, ..., datain 8, we bind the names together using the execute command for that purpose, [bind & datain & vectors ^x].



58. The dataout stream must also be given an external name with the binding command [bind b dataout b results 'x']. The default name for files is the stream name.

59. The source data base must be compiled to convert it into assembly code. (The result of compiling <name>.x is <name>.s.) The assembly code must be converted into object form. (The result of assembling <name>.s is <name>.o.) The object code is then coordinated - an experimental optimisation activity that is not implemented in the distributed version. The communication graph must be compiled, producing the connections (readable) and connections.o files. The resulting object code is loaded into the Pringle emulator. All of these activities are implemented by one make operation [\$m].

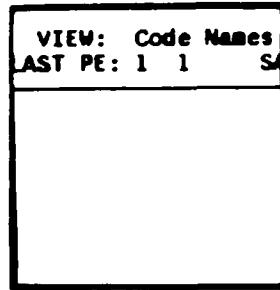
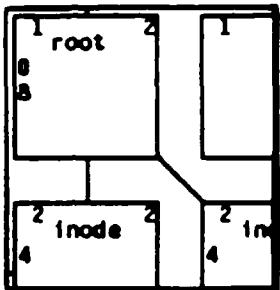
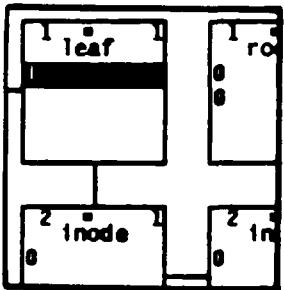
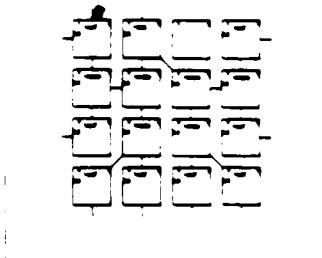
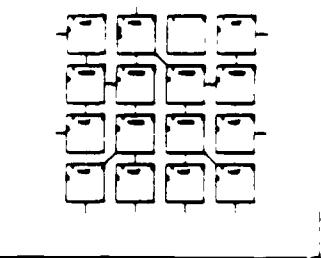
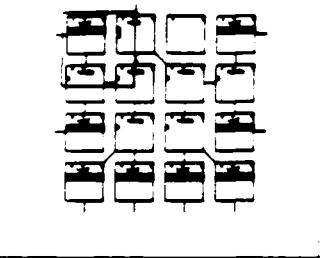
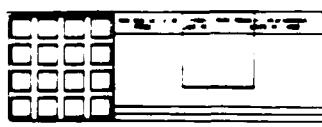
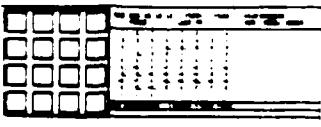
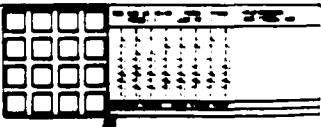
60. Having converted the source database to executable form, having loaded the object code into the emulator and having bound streams to files, it is time to execute the program. It can be done from the Command Request View, but to watch the progress of the execution, we must go to the Trace View [\$t].



61. The Trace View shows a display similar to Code Names View – indeed, the process name is given in each PE box. However, instead of the (actual) parameters being listed after the process name, the values of the trace variables (initially 0) are listed. Each time execution stops, the current values of the traced variables are given. To execute the program until the first event takes place, we use the event command [`^e`].

62. The execution is initiated which can be seen because each PE has an equal sign in the home position showing that it is running. We can execute two more events with [`^e`].

63. The values that have changed are highlighted as the execution proceeds in order to call our attention to the activity. To continue the execution until all PEs are finished and to show a trace of the progress, we type [continue `'x`].



64. Execution continues; when all PEs have halted, we note the correct result (0) in the traced variable in the root.

65. Flushed with success at having gotten the right answer, we try another test to illustrate how successive programs are executed. We will for simplicity use a new file, "doublefile", which is just two copies of vectors. [shell bcpb/usr/poker/lib/Playing Poker /doublefile. ^x]. Since vectors is already formatted we do not have to format doublefile. To change the stream length actual parameter, we move to Code Names View [Sc].

66. We can make a uniform substitution of 20 (the new stream length) for every occurrence of 10 [replace b 10 b 20].

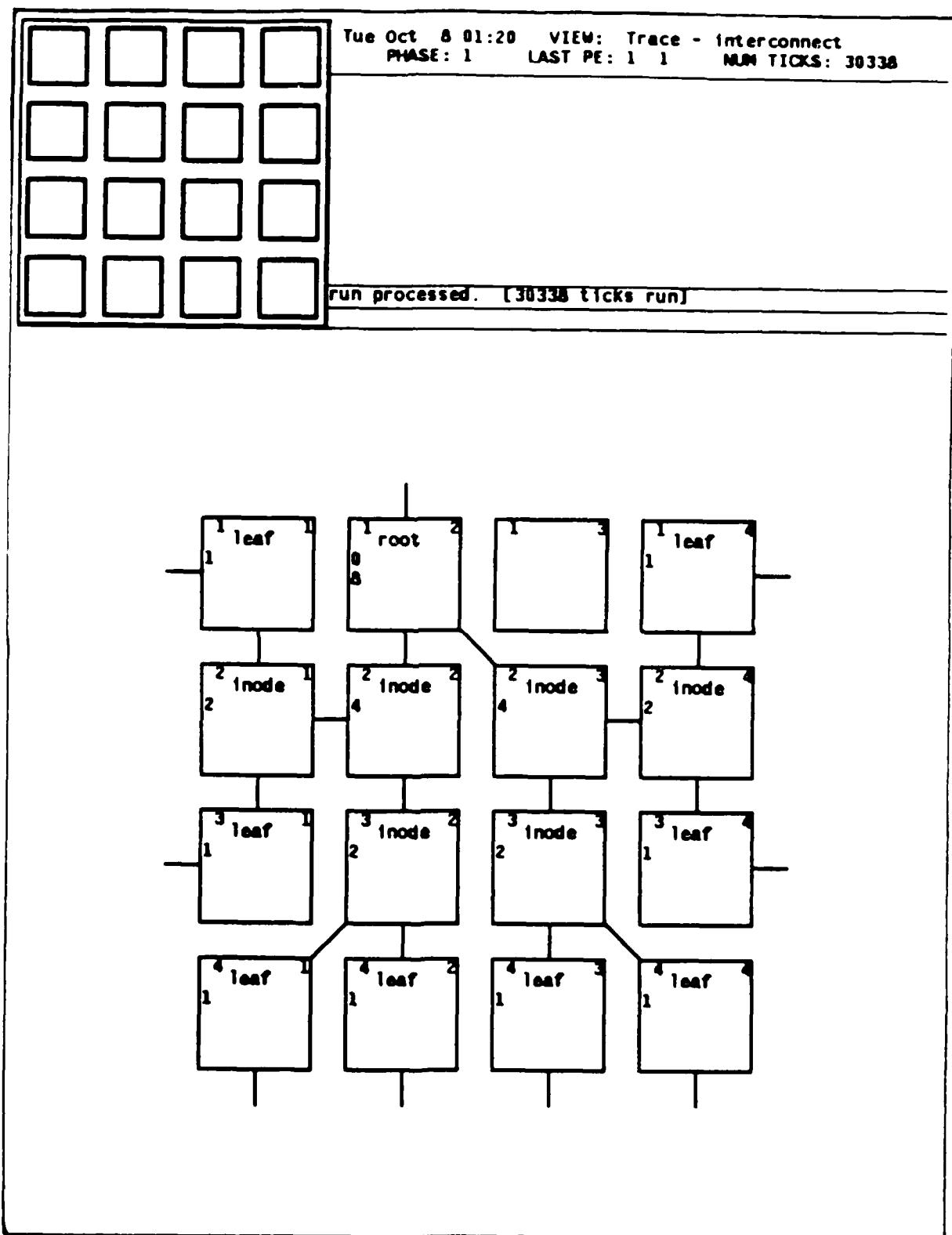
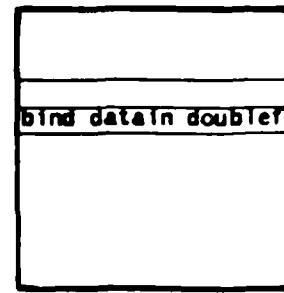
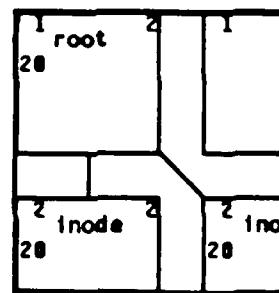
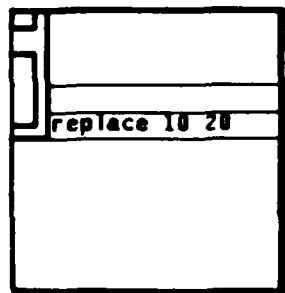
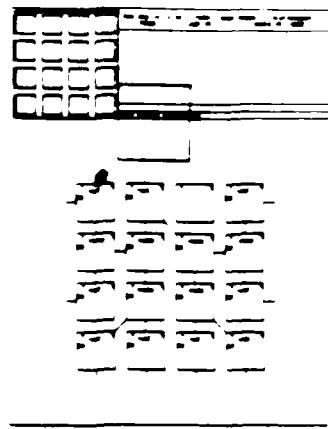
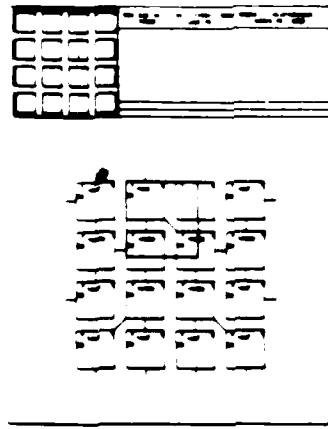
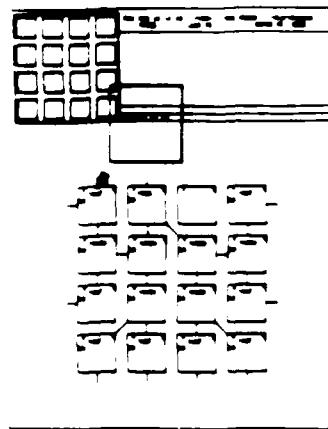
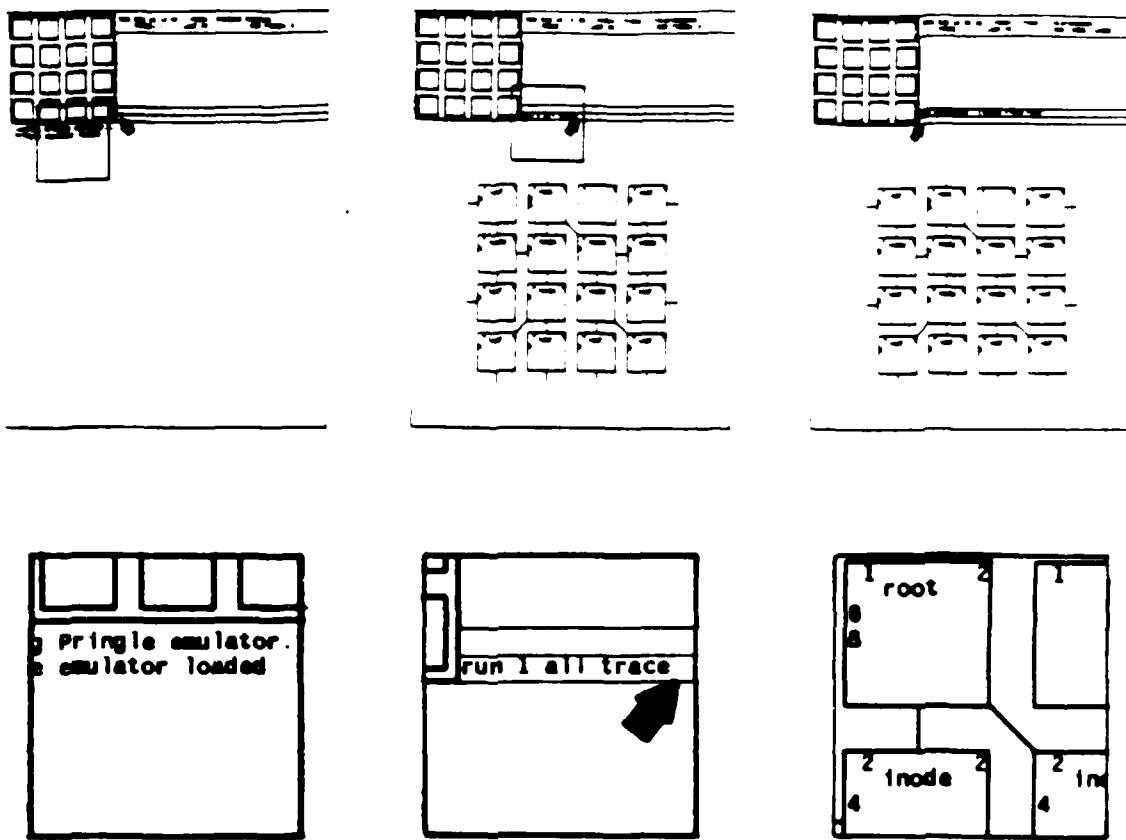


Figure 8.



67. Execution of the command 68. We must also bind the new [^x] implements the substitution. file to the input data streams [bind & datain & doublefile ^x]. Notice that by not changing the file name associated with the output stream, we will simply write over it.

69. We move to Command Request View [Sr] where we load [^l] the current copy of the compiled code.



70. Next we move on to Trace View [**\$t**] where we specify that we want to run (this phase 1) until all of the PEs complete execution [**run b 1 b all b trace ^x**].

71. The emulator execution proceeds autonomously until the condition is realised. (Users who do not wish to run all 52,600 ticks of the example can interrupt execution with **\**).

72. Again, the zero result is achieved. To leave Poker and to save the source state, we exit [**\$^e**].

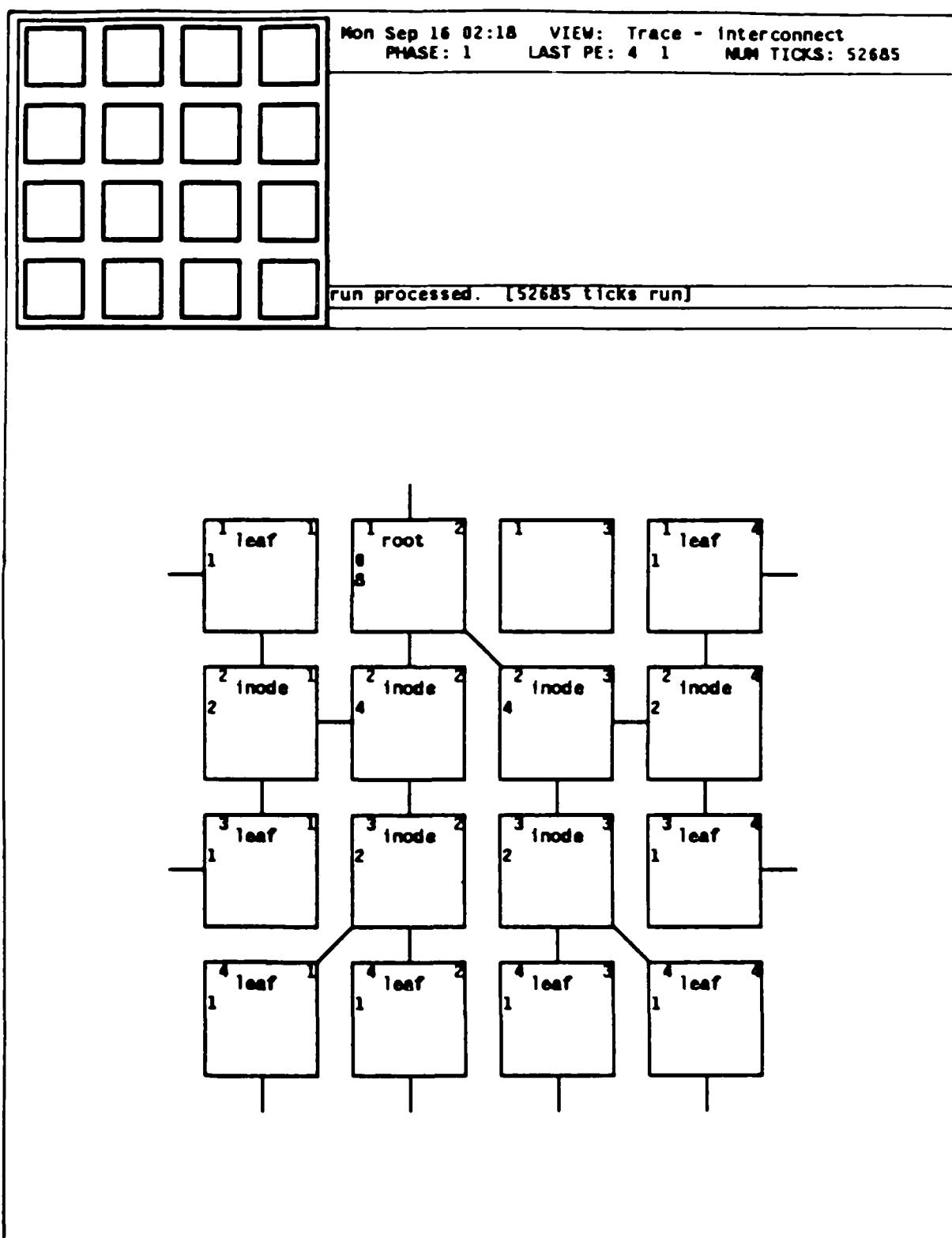


Figure 9.

## Exercise

Although there are many features of Poker that have not been illustrated, enough of the system has been described to enable the reader to solve the following problem.

**Problem** Let the three streams listed below be the coefficients of three polynomials given in increasing powers of  $x$ . Define a "slave" process that receives an  $x$  value from a "master" process, reads in the coefficients, evaluating the polynomial as it does, and returns the result to the master. The master process, which has the three  $x$  values as parameters, sends a value to each slave, reads the result from each slave, adds together the results and outputs the sum as a (one element) stream.

It is suggested that the slave processes have the number of terms (degree + 1) of the polynomial as a parameter.

The three coefficient streams of length 8, 5 and 4, respectively, are assumed in the following example to be:

1.004	5.078	2.953
49.827	0.553	5.167
3.590	13.422	0.875
0.333	9.244	7.754
6.000	1.144	
0.253		
0.096		
1.000		

The points at which the polynomials are to be evaluated are assumed to be, respectively, 1.011, 2.622, 3.14.

As a hint to solving the exercise, we give the Code Names View for a possible solution, as well as the final result (highlighted) of the traced computation.

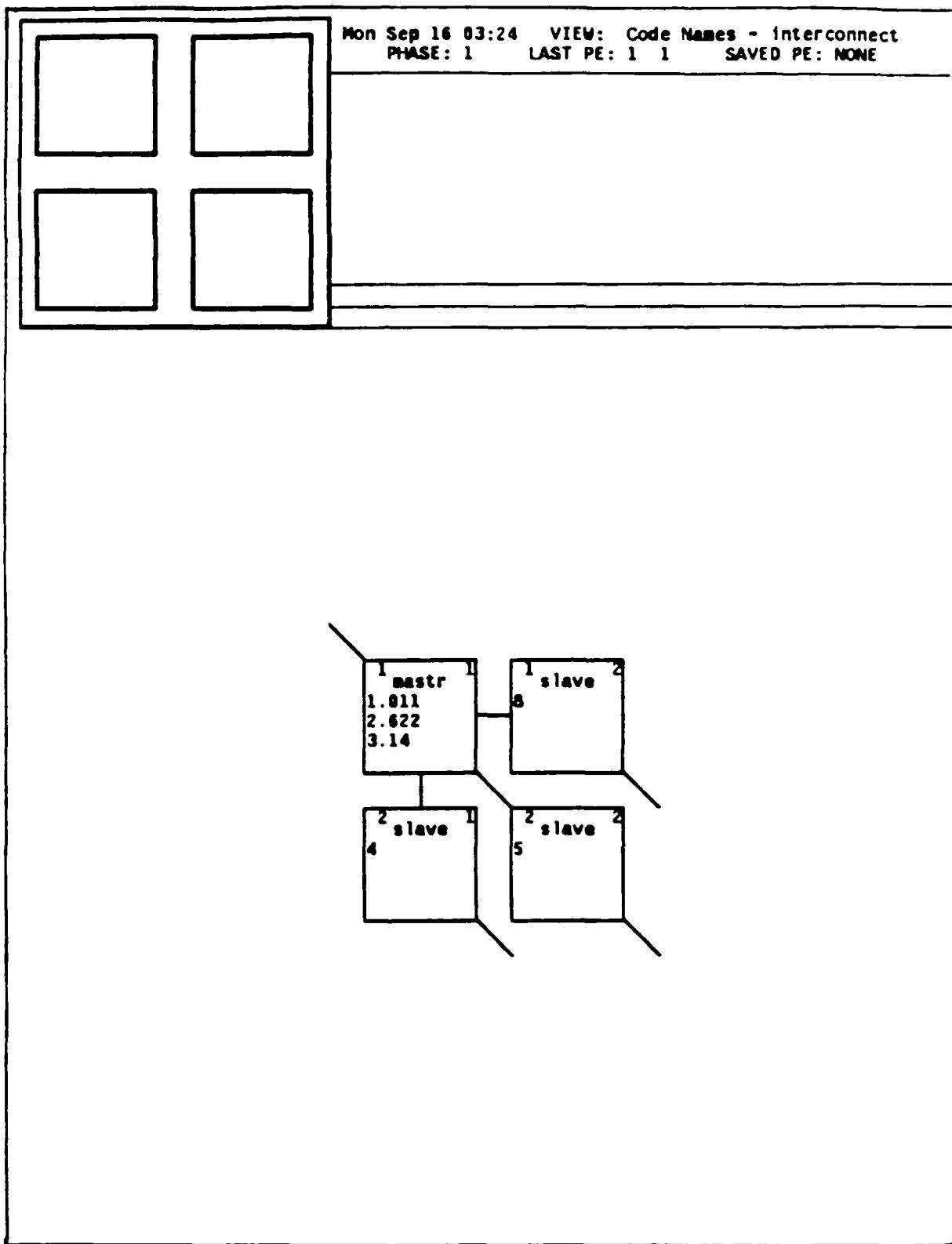


Figure 10.

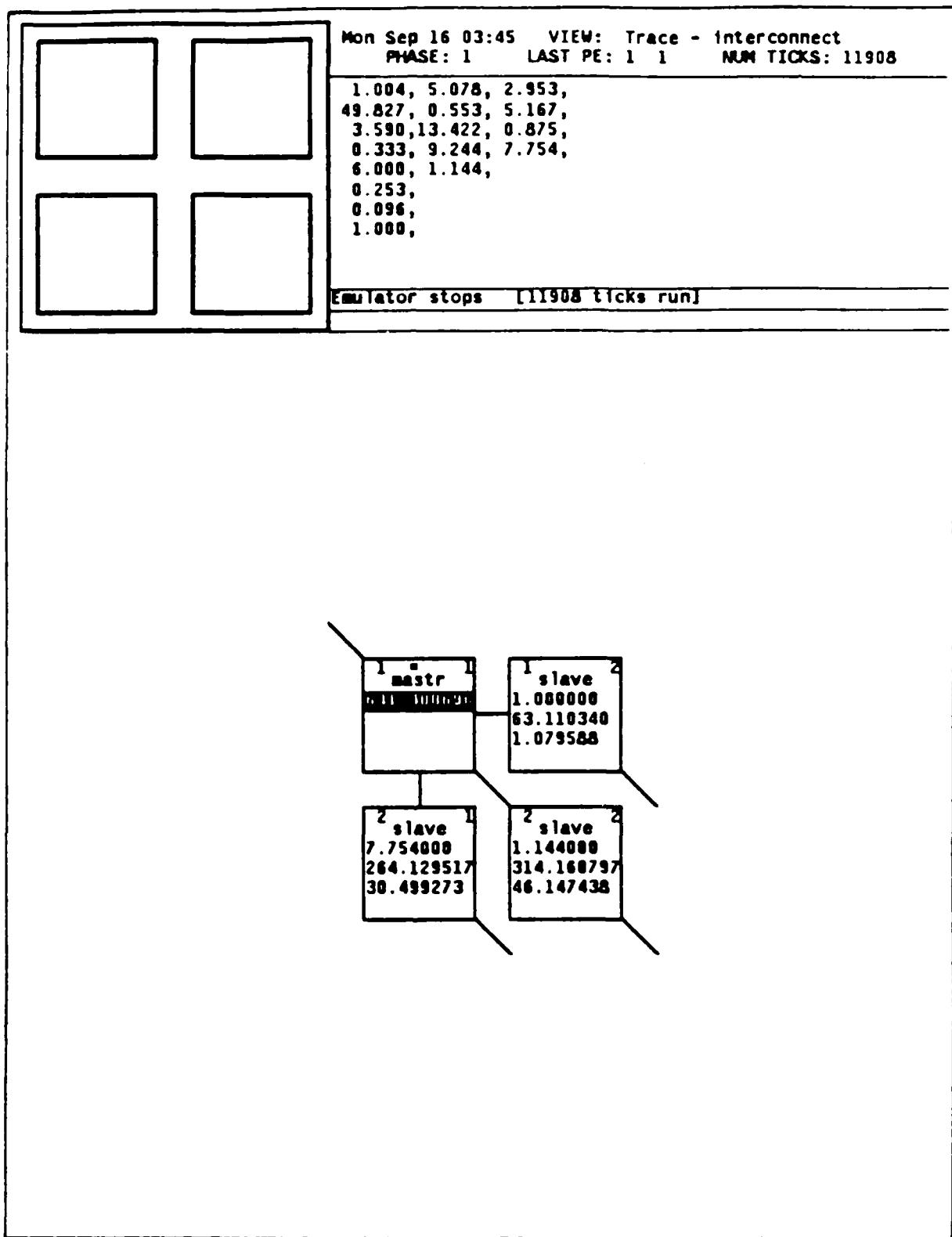


Figure 11.

END  
DATE  
FILED

4-88

DTIC